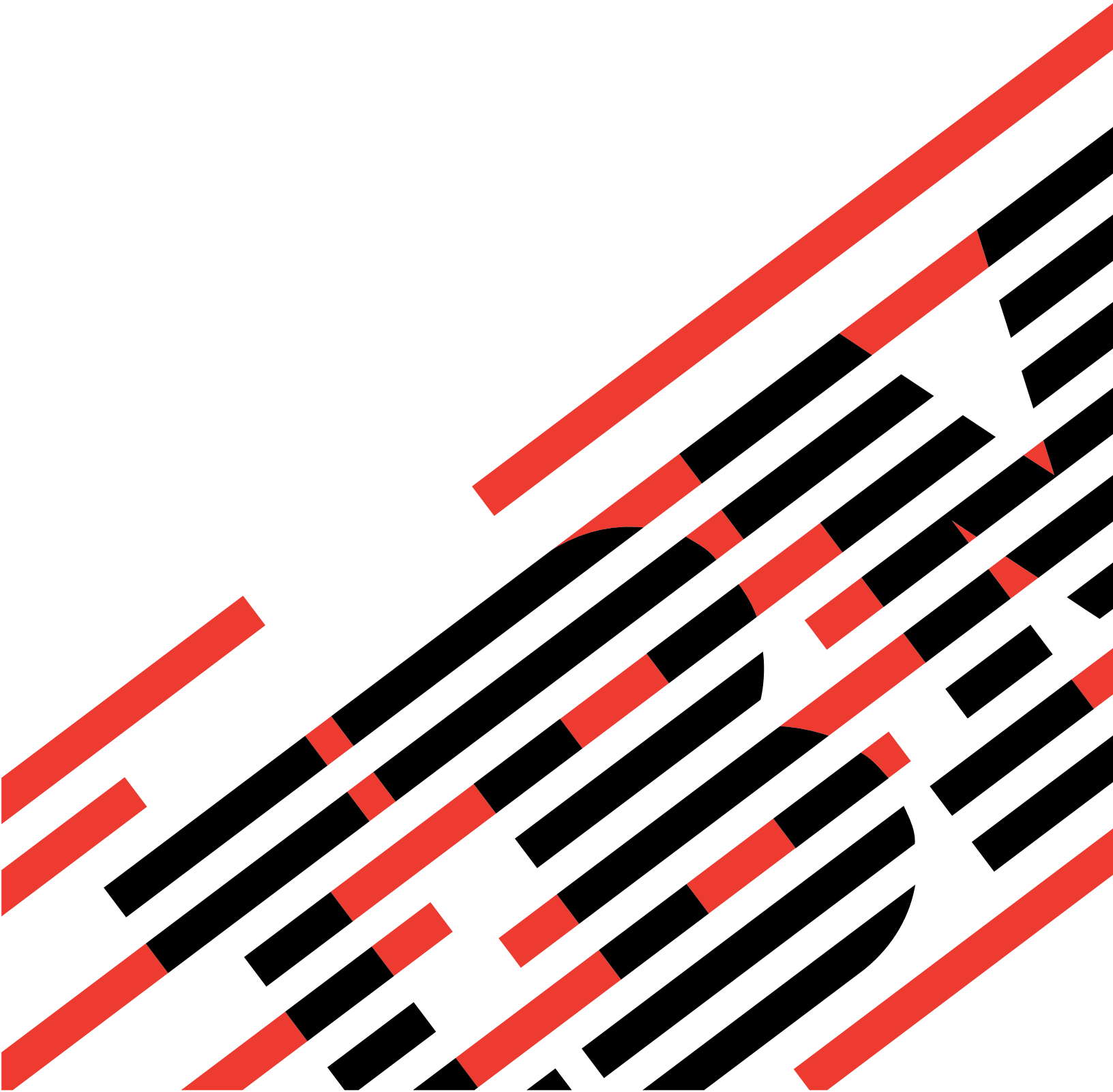
IBM

eserver

iSeries

Machine Interface Instructions APIs

Version 5 Release 3



IBM

@server

iSeries

Machine Interface Instructions APIs

Version 5 Release 3

Note

Before using this information and the product it supports, be sure to read the information in Appendix A, "Notices," on page 1305.

First Edition (May 2004)

This edition applies to version 5, release 3, modification 0 of Operating System/400 (product number 5722-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

Contents

Machine Interface Instructions

Machine interface instructions include:

- "iSeries(TM) Machine Interface"
- ILE C/C++ MI Library Reference

APIs by category

APIs

These are the APIs for this category.

iSeries(TM) Machine Interface

Instruction name

Activate Bound Program (ACTBPGM)

Op Code (Hex)	Operand 1	Operand 2
ACTBPGM2 02DE	Activation defn	Program spec
ACTBPGM 02CE	Activation defn	Program spec

Operand 1: Space pointer.

Operand 2: System pointer or space pointer data object

Bound program access

```
Built-in number for ACTBPGM2 is 658.
ACTBPGM2(
         activation_defn   : address of aggregate
         program_spec      : address of system pointer or
                             address of space pointer(16)
)

OR
Built-in number for ACTBPGM is 416.
ACTBPGM (
         activation_defn   : address of aggregate
         program_spec      : address of system pointer or
                             address of space pointer(16)
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note

It is recommended that you use the ACTBPGM2 instruction which supports 8-byte activation and activation group marks. 4-byte marks can wrap and produce unexpected results.

Description: Activates a bound program or bound service program. The *program spec* operand specifies the target program to be activated, either directly or indirectly.

If the *program spec* operand specifies a space pointer, it must locate a 16-byte aligned structure. The format of the structure is different for the ACTBPGM and ACTBPGM2 instructions.

Format of program spec for ACTBPGM2 instruction:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Program	System pointer	
16	10	Target activation group	UBin(8)	
16	10		Target activation group (Non-Bound program)	Char(8)
24	18	Reserved (binary 0)	Char(24)	
48	30	—- End —-		

Format of program spec for ACTBPGM instruction:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Program	System pointer
16	10	Target activation group	UBin(4)
20	14	Reserved (binary 0)	Char(12)
32	20	—- End —-	

where,

Program is a system pointer to a bound service program to be activated. The service program *must* specify **callers activation group** for its activation group attribute. If the program is not a service program or the activation group attribute is incorrect, then the *invalid operation for program* (hex 2C15) exception is signaled.

Target activation group is the activation group mark of an existing activation group into which the service program is to be activated. If the activation group does not exist, then the *activation group not found* (hex 2C13) exception is signaled.

The service program is eligible to be activated into the target activation group if either of the following conditions hold:

•

- The target activation group's state and the service program state are equal (i.e. either both are *system-state* or both are *user-state*).
- The service program state is *inherit-state* and the target activation group's state is equal to the current *thread execution state*.

If the service program is not eligible to be activated, the *activation access violation* (hex 2C1E) exception is signaled. If the service program is eligible to be activated in the target activation group, activation proceeds as described in *activating the program* below.

If the *program spec* operand specifies a system pointer, it designates the target program to be activated. The target program must be either a bound program or a bound service program, otherwise *invalid operation for program* (hex 2C15) exception is signaled.

Activating the Program: The program activation operation is discussed in detail in the CALLX instruction.

If the target program adopts its owner's user profile, the effect is as if the target program were the most recent invocation on the call stack. This permits the adopted authority of the owner to be applied for purposes of activating dependent service programs. This adoption policy is in effect for the duration of the ACTBPGM operation.

Activation Definition: The *activation defn* must point to a 16-byte aligned area which receives the **activation definition**. The format of the structure is different for the ACTBPGM and ACTBPGM2 instructions.

Format of activation defn for ACTBPGM2 instruction:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Activation definition	Char(48)
0	0		Activation group mark UBin(
0	0		Activation group mark (Non-Bound program)
8	8		Activation mark UBin(
8	8		Activation mark (Non-Bound program)
16	10		Reserved Char(
23	17		Indicators Char(
23	17		Activation status
			0= New activation
			1= Existing activation
23	17		Reserved (binary 0)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
24	18		Reserved (binary 0)	Char(24)
48	30	—- End —-		

Format of activation defn for ACTBPGM instruction:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Activation definition	Char(32)		
0	0		Activation group mark	UBin(4)	
4	4		Activation mark	UBin(4)	
8	8		Reserved	Char(7)	
15	F		Indicators	Char(1)	
15	F		Activation status		Bit 0
			0= New activation		
			1= Existing activation		
15	F		Reserved (binary 0)		Bits 1-7
16	10		Reserved (binary 0)	Char(16)	
32	20	—- End —-			

where,

The **activation group mark** identifies the activation group into which the target program was activated.

The **activation mark** identifies the activation of the program.

The **activation status** indicates whether the operation created a new activation (=0) or found an existing activation (=1).

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

–

– Program referenced by operand 2

– Contexts referenced for address resolution

Lock Enforcement

•

• Materialize

–

– Contexts referenced for address resolution

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

0A Authorization

- 0A01 Unauthorized for Operation

10 Damage Encountered

- 1004 System Object Damage State
- 1005 Authority Verification Terminated Due to Damaged Object
- 1044 Partial System Object Damage

1A Lock State

- 1A01 Invalid Lock State

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2207 Authority Verification Terminated Due to Destroyed Object
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid
- 2403 Pointer Addressing Invalid Object Type

2A Program Creation

2AB5 Observable Information Necessary For Retranslation Not Encapsulated

2C Program Execution

2C12 Activation Group Access Violation

2C15 Invalid Operation for Program

2C1E Activation Access Violation

2C2A Caller Parameter Mask Does Not Match Imported Procedure Parameter Mask

2C2B Invalid Storage Model

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Activate Program (ACTPG)

Op Code (Hex)	Operand 1	Operand 2
0212	Program or static storage frame	Program

Operand 1: Space pointer data object or system pointer.

Operand 2: System pointer.

Bound program access

```
Built-in number for ACTPG is 32.
ACTPG (
       program_or_static_storage_frame   : address of system pointer OR
                                           address of space pointer(16)
       program                           : address of system pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction creates an activation entry for the non-bound *program* specified by operand 2, if it uses static storage. If the *program* specified is of any other type, an *invalid operation for program* (hex 2C15) exception is signaled. No operation is performed for a program which does not require static storage.

Operand 1 receives either a space pointer or system pointer as follows:

•

- If an activation entry is created or an activation entry exists for the program within the target activation group, then a space pointer to the static storage frame is returned. The static storage frame is allocated and initialized according to specifications within the program. The static storage frame is 16-byte aligned and begins with a 64-byte header. The header is not initialized and it is not used by the machine. The header is provided for compatibility with prior machine implementations.
- If the program does not use static storage (hence, no activation entry is created) a copy of the program pointer in operand 2 is returned.

If an attempt is made to activate an already active program then

- the activation mark of the activation entry is changed, and
- the static storage frame is reinitialized

  When the security level machine attribute value is hex 40 and higher, if the thread state at the time this instruction is invoked is user state and an attempt is made to activate a system state program, an *invalid operation for program* (hex 2C15) exception will be signalled.

A space pointer machine object may not be specified for operand 1.

Warning: Temporary Level 3 Header

Authorization Required

•

- Execute
  –
  – Program referenced by operand 2
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2A Program Creation

2AB5 Observable Information Necessary For Retranslation Not Encapsulated

2C Program Execution

2C15 Invalid Operation for Program

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Add Logical Character (ADDLC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
ADDLC 1023		Sum	Addend 1	Addend 2	
ADDLCI 1823	Indicator options	Sum	Addend 1	Addend 2	Indicator targets
ADDLCB 1C23	Branch options	Sum	Addend 1	Addend 2	Branch targets

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4-7:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
ADDLCS 1123		Sum/Addend 1	Addend 2	
ADDLCIS 1923	Indicator options	Sum/Addend 1	Addend 2	Indicator targets
ADDLCBS 1D23	Branch options	Sum/Addend 1	Addend 2	Branch targets

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The unsigned binary value of the addend 1 operand is added to the unsigned binary value of the addend 2 operand and the result is placed in the sum operand.

If the short form is not used and if neither source operand is an immediate value, then operands 2 and 3 must be the same length. The length can be a maximum of 256 bytes. In the case that the short form is not used and operand 2 or 3 is an immediate operand, it is treated as a character value and extended on the right with hex 00 bytes to match the length of the other operand.

The addition operation is performed according to the rules of algebra. The result value is then placed (left-adjusted) in the receiver operand with truncating or padding taking place on the right. The pad value used in this instruction is a byte value of hex 00.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Resultant Conditions: The logical sum of the character scalar operands is:

-
- Zero with no carry out of the leftmost bit position
- Not-zero with no carry
- Zero with carry
- Not-zero with carry.

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Add Numeric (ADDN)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
ADDN 1043		Sum	Addend	Augend	
ADDNR 1243		Sum	Addend	Augend	
ADDNI 1843	Indicator options	Sum	Addend	Augend	Indicator targets
ADDNIR 1A43	Indicator options	Sum	Addend	Augend	Indicator targets
ADDNB 1C43	Branch options	Sum	Addend	Augend	Branch targets
ADDNBR 1E43	Branch options	Sum	Addend	Augend	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4-7:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
ADDNS 1143		Sum/Addend	Augend	
ADDNSR 1343		Sum/Addend	Augend	
ADDNIS 1943	Indicator options	Sum/Addend	Augend	Indicator targets
ADDNISR 1B43	Indicator options	Sum/Addend	Augend	Indicator targets
ADDNBS 1D43	Branch options	Sum/Addend	Augend	Branch targets
ADDNBSR 1F43	Branch options	Sum/Addend	Augend	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Caution: If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used or whenever the *assume coincident operand overlap* attribute has been specified in the program template. If the *assume coincident operand overlap* attribute has not been specified in the program template and indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Description: The *sum* is the result of adding the *addend* and *augend*.

Operands can have floating-point, packed or zoned decimal, signed or unsigned binary type.

Source operands are the *addend* and *augend.* The receiver operand is the *sum.*

If operands are not of the same type, addends are converted according to the following rules:

1. If any one of the operands has floating point type, addends are converted to floating point type.
2. Otherwise, if any one of the operands has zoned or packed decimal type, addends are converted to packed decimal.
3. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Addend and *augend* are added according to their type. Floating point operands are added using floating point addition. Packed decimal addends are added using packed decimal addition. Unsigned binary addition is used with unsigned addends. Signed binary addends are added using two's complement binary addition.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary additions execute faster than either packed decimal or floating point additions.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

For a decimal operation, alignment of the assumed decimal point takes place by padding with 0's on the right end of the *addend* with lesser precision.

Floating-point addition uses exponent comparison and significand addition. Alignment of the binary point is performed, if necessary, by shifting the significand of the value with the smaller exponent to the right. The exponent is increased by one for each binary digit shifted until the two exponents agree.

The operation uses the lengths and the precision of the source and receiver operands to calculate accurate results. Operations performed in decimal are limited to 31 decimal digits in the intermediate result.

The addition operation is performed according to the rules of algebra.

The result of the operation is copied into the *sum* operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the *sum,* aligned at the assumed decimal point of the *sum* operand, or both before being copied. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations outlined in the Arithmetic Operations. If nonzero digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

When the target of the instruction is signed or unsigned binary size, exceptions can be suppressed.

For the optional round form of the instruction, specification of a floating-point receiver operand is invalid.

For fixed-point operations, if nonzero digits are truncated off the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For floating-point operations involving a fixed-point receiver field, if nonzero digits would be truncated off the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point sum, if the exponent of the resultant value is either too large or too small to be represented in the sum field, the *floating-point overflow* (hex 0C06) exception and *floating-point underflow* (hex 0C07) exception are signaled, respectively.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Conditions:

-
- Positive - The algebraic value of the numeric scalar sum operand is positive.
- Negative - The algebraic value of the numeric scalar sum operand is negative.
- Zero - The algebraic value of the numeric scalar sum operand is zero.
- Unordered - The value assigned a floating-point sum operand is NaN.

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C03 Decimal Point Alignment

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Add Space Pointer (ADDSPP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0083	Receiver pointer	Source pointer	Increment

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Binary scalar.

Description: This instruction adds a signed or unsigned binary value to the offset of a space pointer. The value of the binary scalar represented by operand 3 is added to the space address contained in the space pointer specified by operand 2, and the result is stored in the space pointer identified by operand 1. I.e.

```
Operand 1 = Operand 2 + Operand 3
```

Operand 3 can have a positive or negative value. The space that the pointer is addressing is not changed by the instruction.

Operand 2 must contain a space pointer; otherwise, a *pointer type invalid* (hex 2402) exception is signaled.

When the addressability in the space pointer is modified, the instruction signals a *space addressing violation* (hex 0601) exception when one of the following conditions occurs, for any space except teraspace:

-
- The space address to be stored in the pointer has a negative offset value.
- The offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated.

If the exception is signaled by this instruction for one of these reasons, the pointer is not modified by the instruction.

In contrast, when modifying the addressability of a space pointer to teraspace, if the address computed either overflows or underflows the offset, the result is wrapped back within teraspace and no exception is signalled. However, since the size of teraspace and thus the size of the offset portion of a teraspace address is implementation-dependent, the wrapped result may vary between machine implementations.

Attempts to use a pointer whose offset value lies: between the currently allocated extent of the space and the maximum allocatable extent of the space, or whose offset is outside all teraspace allocations, cause the *space addressing violation* (hex 0601) exception to be signaled.

The *object destroyed* (hex 2202) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 and operand 2 are space pointer machine objects. This occurs when operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition that existed for operand 2. The appropriate exception condition will be signaled for either pointer when a subsequent attempt is made to reference the space data that the pointer addresses.

Warning: Temporary Level 3 Header

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Allocate Activation Group-Based Heap Space Storage (ALCHSS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03B3	Space allocation	Heap identifier	Size of space allocation

Operand 1: Space pointer.

Operand 2: Binary(4) scalar or null.

Operand 3: Binary(4) scalar.

Bound program access
Built-in number for ALCHSS is 111. ALCHSS (heap_identifier : signed binary(4) OR unsigned binary(4) OR null operand size_of_space_allocation : signed binary(4)) : space pointer(16) to a space allocation The *heap_identifier* and *size_of_space_allocation* operands correspond to operands 2 and 3 on the ALCHSS operation; the return value corresponds to operand 1.

Note: The term ″heap space″ in this instruction refers to an ″activation group-based heap space″.

Description: A heap space storage allocation of at least the size indicated by operand 3 is provided from the heap space indicated by operand 2. The operand 1 space pointer is set to address the first byte of the allocation which will begin on a boundary at least as great as the minimum boundary specified when the heap space was created.

Each allocation associated with a heap space provides a continuum of contiguously addressable bytes. Individual allocations within a heap space have no addressability affinity with each other. The contents of the heap space allocation are unpredictable unless initialization of heap allocations was specified when the heap space was created.

The maximum single allocation allowed is determined by the maximum single allocation size specified when the heap space was created. The maximum single allocation possible is (16M - 1 page) bytes. To determine the current page size use option hex 12 of the MATRMD instruction. If a user attempts to request a space allocation size of zero or greater than the maximum allocation, an *invalid size request* (hex 4504) exception will be signaled.

It is the responsibility of the using program to see that only the amount of heap space storage requested is used. Reference to heap space storage outside the bounds of the requested space will produce unpredictable results. The exact address returned must be supplied to the Free Activation Group-Based Heap Space Storage (FREHSS) instruction when the user has completed use of the heap space storage.

A default heap space (heap identifier value of 0) is automatically available in each activation group without issuing a Create Activation Group-Based Heap Space (CRTHS) instruction. The default heap space is created when the first Allocate Activation Group-Based Heap Space is issued against the default heap space. When operand 2 is null, the default heap space (heap identifier of 0) provides the allocation.

The machine supplied attributes of the default heap space are as follows:

-
- Maximum single allocation size is (16M - 1 page) bytes.
- Minimum boundary requirement is a 16 byte boundary.
- The creation size advisory is 4KB unless the size of the allocation request dictates a larger creation size be used.
- The extension size advisory is 4KB unless the size of the allocation request dictates a larger extension size be used.
- Domain is determined from the state of the program issuing the instruction.
- Normal allocation strategy.
- A heap space mark is not allowed.
- The transfer size is 1 page.
- The process access group membership advisory value is taken from the activation group.

- Heap space storage allocations are not initialized to the allocation value.
- Heap space storage allocations are not overwritten to the freed value after being freed.

Neither operand 2 nor 3 is modified by the instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

45 Heap Space

4501 Invalid Heap Identifier

4503 Heap Space Full

4504 Invalid Size Request

4505 Heap Space Destroyed

4506 Invalid Heap Space Condition

And (AND)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
AND 1093		Receiver	Source 1	Source 2	
ANDI 1893	Indicator options	Receiver	Source 1	Source 2	Indicator targets
ANDB 1C93	Branch options	Receiver	Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character scalar or numeric scalar.

Operand 4-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-4]
ANDS 1193		Receiver/Source 1	Source 2	
ANDIS 1993	Indicator options	Receiver/Source 1	Source 2	Indicator targets
ANDBS 1D93	Branch options	Receiver/Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3-4:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Boolean **and** operation is performed on the string values in the source operands. The resulting string is placed in the receiver operand. The operands may be character or numeric scalars. They are both interpreted as bit strings. Substringing is supported for both character and numeric operands.

The length of the operation is equal to the length of the longer of the two source operands. The shorter of the two operands is logically padded on the right with hex 00 values. This assigns hex 00 values to the results for those bytes that correspond to the excess bytes of the longer operand.

The bit values of the result are determined as follows:

Source 1 Bit	Source 2 Bit	Result Bit
0	0	0
0	1	0
1	0	0
1	1	1

The result value is then placed (left-adjusted) in the receiver operand with truncating or padding taking place on the right. The pad value used in this instruction is a byte value of hex 00.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1, 2, and 3. The effect of specifying a null substring reference for either or both of the source operands is that the result is all zero and the instruction's resultant condition is *zero*. When a null substring reference is specified for the receiver, a result is not set and the instruction's resultant condition is *zero* regardless of the values of the source operands.

When the receiver operand is a numeric variable scalar, it is possible that the result produced will not be a valid value for the numeric type. This can occur due to padding with hex 00, due to truncation, or due to the resultant bit string produced by the instruction. The instruction completes normally and signals no exceptions for these conditions.

Resultant Conditions:

•

- Zero - The bit value for the bits of the scalar receiver operand is either all zero or a null substring reference is specified for the receiver.
- Not zero - The bit value for the bits of the scalar receiver operand is not all zero.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

And Complemented String (ANDCSTR)

Bound program access

```
Built-in number for ANDCSTR is 463.
ANDCSTR (
         receiver_string        : address of aggregate(*)
         first_source_string    : address of aggregate(*)
         second_source_string   : address of aggregate(*)
         string_length          : unsigned binary(4,8) value which specifies
                                  the length of the three strings
)
```

Description: Each byte value of the *first source string*, for the number of bytes indicated by *string length*, is logically **and**ed with the logical complement of the corresponding byte value of the *second source string*, on a bit-by-bit basis. The results are placed in the *receiver string*. If the strings overlap in storage, predictable results occur only if the overlap is fully coincident.

If the space(s) indicated by the three addresses are not long enough to contain the number of bytes indicated by *string length*, a *space addressing violation* (hex 0601) is signalled. Partial results in this case are unpredictable.

An example of the AND-COMPLEMENT operation, where the two operands each have length of one byte, follows:

```
First operand value:           01101001
Second operand value:          10010001
Second operand complemented:   01101110
Final result of first operand
ANDed with the complement of
the second operand:            01101000
```

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

AND String (ANDSTR)

Bound program access

```
Built-in number for ANDSTR is 450.
ANDSTR (
        receiver_string        : address of aggregate(*)
        first_source_string    : address of aggregate(*)
        second_source_string   : address of aggregate(*)
        string_length          : unsigned binary(4,8) value which specifies
                                 the length of the three strings
)
```

Description: Each byte value of the *first source string*, for the number of bytes indicated by operand 4, is logically **and**ed with the corresponding byte value of the *second source string*, on a bit-by-bit basis. The results are placed in the *receiver string*. If the strings overlap in storage, predictable results occur only if the overlap is fully coincident.

If the space(s) indicated by the three addresses are not long enough to contain the number of bytes indicated by *string length*, a *space addressing violation* (hex 0601) is signalled. Partial results in this case are unpredictable.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Arc Cosine (ACOS)

Bound program access

`Built-in number for ACOS is 401.`
`ACOS (`
`  source  : floating point(8) value`
`) :  floating point(8) value which is the arc cosine of the`
`  source value`

Description: The arc cosine of the numeric value of the *source* operand is computed and the result (in radians) is returned.

The result is in the range:

```
0 <= ACOS(source) <= pi
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Arc Sine (ASIN)

Bound program access

`Built-in number for ASIN is 399.`
`ASIN (`
`  source  : floating point(8) value`
`) :  floating point(8) value which is the arc sine of the source value`

Description: The arc sine of the numeric value of the *source* operand is computed and the result (in radians) is returned.

The result is in the range:

```
-pi/2 <= ASIN(source) <= +pi/2
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Arc Tangent (ATAN)

Bound program access

`Built-in number for ATAN is 403.`
`ATAN (`
`      source   : floating point(8) value`
`) :   floating point(8) value which is the arc tangent of the source value`

Description: The arc tangent of the numeric value of the *source* operand is computed and the result (in radians) is returned.

The result is in the range:

```
-pi/2 <= ATAN(source) <= +pi/2
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Arc Tangent Hyperbolic (ATANH)

Bound program access
Built-in number for ATANH is 410. ATANH (source : floating point(8) value) : floating point(8) value which is the arc tangent hyperbolic of the source value

Description: The inverse of the tangent hyperbolic of the numeric value of the *source* operand is computed and the result (in radians) is returned.

The result is in the range:

```
-infinity <= ATANH(source) <= +infinity
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Atomic Add (ATMCADD)

Bound program access

```
Built-in number for ATMCADD4 is 671.
ATMCADD4 (
          op1   : address of a signed binary(4) value  (has
                    alignment restrictions
                    - see description below)
          op2   : signed binary(4) value
) : signed binary(4)

Built-in number for ATMCADD8 is 672.
ATMCADD8 (
          op1   : address of a signed binary(8) value  (has
                    alignment restrictions
                    - see description below)
          op2   : signed binary(8) value
) : signed binary(8)
```

Description: Atomically increments the value pointed to by *op1* by the value *op2*. Returns the original value pointed to by *op1*.

The value pointed to by *op1* and the *op2* value must have the same length. Failure to have the operands the same length will not be detected and the results of the instruction are undefined when this occurs.

The first operand must be aligned based on its length:

•
- four byte length - 4-byte aligned
- eight byte length - 8-byte aligned

Failure to have the first operand aligned properly will not be detected, but the results of the instruction are undefined when this occurs.

The arithmetic performed by this instruction will not signal any exceptions.

This operation is useful when a variable is shared between two or more threads. When updating such a variable, it is important to make sure that the entire operation is performed atomically (not interruptible). See “Atomicity” on page 1275 for additional information.

The primary purpose of this instruction is to manipulate a variable which is shared by two or more threads, but this instruction does not synchronize storage. When sharing more than one variable between multiple threads or processes, be aware of storage synchronization issues. See Storage Synchronization Concepts for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•
- None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2C Program Execution

- 2C04 Branch Target Invalid

36 Space Management

- 3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Atomic And (ATMCAND)

Bound program access

```
Built-in number for ATMCAND4 is 673.
ATMCAND4 (
          op1   : address of a unsigned binary(4) value  (has
                    alignment restrictions
                    - see description below)
          mask  : unsigned binary(4) value
) : unsigned binary(4)
Built-in number for ATMCAND8 is 674.
ATMCAND8 (
          op1   : address of a unsigned binary(8) value  (has
                    alignment restrictions
                    - see description below)
          mask  : unsigned binary(8) value
) : unsigned binary(8)
```

Description: Sets bits in the value pointed to by *op1*, according to a bit mask, in a single atomic operation. The bits in the value pointed to by *op1* that correspond to the zero bits in *mask* are set to 0. The bits in the value pointed to by *op1* that correspond to the one bits in *mask* are not modified. Returns the original value pointed to by *op1*.

The updated bit values for the storage pointed to by *op1* are determined as follows:

Original op1 Bit	Mask Bit	Resulting op1 Bit
0	0	0
0	1	0
1	0	0
1	1	1

The value pointed to by *op1* and the *mask* value must have the same length. Failure to have the operands the same length will not be detected and the results of the instruction are undefined when this occurs.

The first operand must be aligned based on its length:

-
- four byte length - 4-byte aligned
- eight byte length - 8-byte aligned

Failure to have the first operand aligned properly will not be detected, but the results of the instruction are undefined when this occurs.

This operation is useful when a variable containing bit flags is shared between two or more threads. When updating such a variable, it is important to make sure that the entire operation is performed atomically (not interruptible). See “Atomicity” on page 1275 for additional information.

The primary purpose of this instruction is to manipulate a variable which is shared by two or more threads, but this instruction does not synchronize storage. When sharing more than one variable between multiple threads or processes, be aware of storage synchronization issues. See Storage Synchronization Concepts for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Atomic Or (ATMCOR)

Bound program access

```
Built-in number for ATMCOR4 is 675.
ATMCOR4 (
          op1   : address of a unsigned binary(4) value  (has
                  alignment restrictions
                  - see description below)
          mask  : unsigned binary(4) value
) : unsigned binary(4)

Built-in number for ATMCOR8 is 676.
ATMCOR8 (
          op1   : address of a unsigned binary(8) value  (has
                  alignment restrictions
                  - see description below)
          mask  : unsigned binary(8) value
) : unsigned binary(8)
```

Description: Sets bits in the value pointed to by *op1*, according to a bit mask, in a single atomic operation. The bits in the value pointed to by *op1* that correspond to the one bits in *mask* are set to 1. The bits in the value pointed to by *op1* that correspond to the zero bits in *mask* are not modified.

The updated bit values for the storage pointed to by *op1* are determined as follows:

Original op1 Bit	Mask Bit	Resulting op1 Bit
0	0	0
0	1	1
1	0	1
1	1	1

The value pointed to by *op1* and the *mask* value must have the same length. Failure to have the operands the same length will not be detected and the results of the instruction are undefined when this occurs.

The first operand must be aligned based on its length:

•
- four byte length - 4-byte aligned
- eight byte length - 8-byte aligned

Failure to have the first operand aligned properly will not be detected, but the results of the instruction are undefined when this occurs.

This operation is useful when a variable containing bit flags is shared between two or more threads. When updating such a variable, it is important to make sure that the entire operation is performed atomically (not interruptible). See “Atomicity” on page 1275 for additional information.

The primary purpose of this instruction is to manipulate a variable which is shared by two or more threads, but this instruction does not synchronize storage. When sharing more than one variable between multiple threads or processes, be aware of storage synchronization issues. See Storage Synchronization Concepts for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Branch (B)

Op Code (Hex)	Operand 1
1011	Branch target

Operand 1: Instruction number, relative instruction number, branch point, instruction pointer, or instruction definition list element.

Description: Control is unconditionally transferred to the instruction indicated in the branch target operand. The instruction number indicated by the branch target operand must be within the instruction stream containing the branch instruction.

The branch target may be an element of an array of instruction pointers or an element of an instruction definition list. The specific element can be identified by using a compound subscript operand.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Call External (CALLX)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0283	Program to be called or call template	Argument list	Return list

Operand 1: System pointer or space pointer data object.

Operand 2: Operand list or null.

Operand 3: Instruction definition list or null.

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The instruction preserves the calling invocation and passes control to either the *program entry procedure* of a bound program or the external entry point of a non-bound program. If operand 1 specifies a Java(TM) program or a bound program which does not contain a *program entry procedure*, an *invalid operation for program* (hex 2C15) exception is signaled.

Operand 1 may be specified as a system pointer which directly addresses the program that is to receive control or as a space pointer to a call template which identifies the program to receive control. Specifying a template allows for additional controls over how the specified program is to be invoked. The format of the *call template* is the following:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Call options	Char(4)	
0	0		Suppress adopted user profiles	Bit 0
			0 = No	
			1 = Yes	
0	0		Reserved (binary 0)	Bits 1-30
0	0		Force thread state to user state for call	Bit 31

Offset Dec	Hex	Field Name		Data Type and Length
			0 =	No
			1 =	Yes
4	4	Reserved (binary 0)		Char(12)
16	10	Program to be called		System pointer
32	20	—- End —-		

The **suppress adopted user profiles** call option specifies whether or not the program adopted and propagated user profiles which may be serving as sources of authority to the thread are to be suppressed from supplying authority to the new invocation. Specifying *yes* causes the propagation of adopted user profiles to be stopped as of the calling invocation, thereby, not allowing the called invocation to benefit from their authority. Specifying *no* allows the normal propagation of adopted and propagated user profiles to occur. The called program may adopt its owning user profile, if necessary, to supplement the authority available to its invocation.

The **force thread state to user state** option specifies whether or not the call changes the state of the thread to user state.

Operand 2 specifies an operand list that identifies the arguments to be passed to the invocation entry to be called. If operand 2 is null, no arguments are passed by the instruction. An *argument list length violation* (hex 0802) exception is signaled if the number of arguments passed does not correspond to the number required by the parameter list of the target program.

An *unsupported space use* (hex 0607) exception is signalled if this call would pass a parameter stored in teraspace to a program which is not teraspace capable. To be teraspace capable, a non-bound program must be created as teraspace capable or a bound program must be created with a teraspace capable program entry procedure.

Operand 3 specifies an instruction definition list (IDL) that identifies the instruction number(s) of alternate return points within the calling invocation. A Return External instruction in an invocation immediately subordinate to the calling invocation can indirectly reference a specific entry in the IDL to cause a return of control to the instruction associated with the referenced IDL entry. If operand 3 is null, then the calling invocation has no alternate return points associated with the call. If operand 3 is not null and operand 1 specifies a bound program, an *invalid operation for program* (hex 2C15) exception is signaled.

Common Program Call Processing: The details of processing differ for non-bound and bound programs. The following outlines the common steps.

1. A check is made to determine if the caller has authority to invoke the program and that the object is indeed a program object. The specified program must be either a bound program that contains a *program entry procedure* or a non-bound program.
2. The activation group in which the program is to be run is located or created if it doesn't exist.
3. If the program requires an activation entry and it is not already active within the appropriate activation group, it is activated. Bound programs always require an activation; non-bound programs require an activation only if they use static storage. The *invocation count* of a newly created activation is set to 1 while the *invocation count* of an existing activation is incremented by 1.
4. The invocation created for the target program has the following attributes (as would be reported via the Materialize Invocation Attributes (MATINVAT) instruction.)
   -

- the *invocation mark* is at least one higher than any previous invocation within the thread. The *invocation mark* value is generated from the *thread mark counter* and is unique within the thread. There is no relationship between the values of the invocation mark and the marks of the *activation* or *activation group* associated with the invocation.
- the *invocation number* is one greater than the invocation number of the calling invocation. This is merely a measure of the depth of the call-stack.
- the *invocation type* is hex 01 to indicate a CALLX invocation.
- the *invocation number* is the same as the invocation number of the transferring invocation.
- the *invocation type* is hex 02 to indicate a XCTL type of invocation.

5. The automatic storage frame (ASF), if required, is allocated on a 16-byte boundary.
6. Control is transferred to the program entry procedure (or external entry point) of the program.
7. Normal flow-of-control resumes at the instruction following the program call instruction after a return from the program.
8. Normal flow-of-control resumes at the instruction following the caller of the program issuing the XCTL instruction.

The details of locating the target activation group and activating the program differ depending upon the model of the program.

Bound Program: A bound program is activated and run in an activation group specified by program attributes. There are two logical steps involved:

-
- locate the existing, or create a new activation group for the program
- locate an existing, or create a new activation entry for the program within the activation group

After locating the activation entry for the program, control is passed to the program entry procedure for the program. If required, the activation group is destroyed when the invocation for the program entry procedure is destroyed.

Non-bound Program: The automatic storage frame begins with a 64 byte header. If the program defines no automatic data items the frame consists solely of the 64-byte header, otherwise the automatic storage items are located immediately following the header. In prior releases of the machine, this header contained invocation information which is now available via the Materialize Invocation Attributes (MATINVAT) instruction. This header is not initialized and the contents of the header are not used by the machine. (The space is allocated merely to provide for compatibility with prior implementations of the machine.) The *update PASA stack* program attribute, supported in prior implementations of the machine, is no longer meaningful and is ignored, if specified as an attribute of the program.

Following the allocation and initialization of the invocation entry, control is passed to the invoked program.

Warning: Temporary Level 3 Header

Authorization Required

-
- Execute
  -
  - Program referenced by operand 1
  - Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

0802 Argument List Length Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2A Program Creation

2AB5 Observable Information Necessary For Retranslation Not Encapsulated

2C Program Execution

2C15 Invalid Operation for Program

2C1D Automatic Storage Overflow

2C1E Activation Access Violation

2C1F Program Signature Violation

2C20 Static Storage Overflow

2C21 Program Import Invalid

2C22 Data Reference Invalid

2C23 Imported Object Invalid

2C24 Activation Group Export Conflict

2C25 Import Not Found

2C2A Caller Parameter Mask Does Not Match Imported Procedure Parameter Mask

2C2B Invalid Storage Model

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Call Internal (CALLI)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0293	Internal entry point	Argument list	Return target

Operand 1: Internal entry point.

Operand 2: Operand list or null.

Operand 3: Instruction pointer.

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The *internal entry point* specified by operand 1 is located in the same invocation from which the Call Internal instruction is executed. A subinvocation is defined and execution control is transferred to the first instruction associated with the *internal entry point*. The instruction does not cause a new invocation to be established. Therefore, there is no allocation of objects and instructions in the subinvocation have access to all invocation objects.

Operand 2 specifies an operand list that identifies the arguments to be passed to the subinvocation. If operand 2 is null, no arguments are passed. After an argument has been passed on a Call Internal instruction, the corresponding parameter may be referenced. This causes an indirect reference to the storage area located by the argument. This mapping exists until the parameter is assigned a new mapping based on a subsequent Call Internal instruction. A reference to an internal parameter before its being assigned an argument mapping causes a *parameter reference violation* (hex 0801) exception to be signaled.

Operand 3 specifies an instruction pointer that identifies the pointer into which the machine places addressability to the instruction immediately following the Call Internal instruction. A branch instruction in the called subinvocation can directly reference this instruction pointer to cause control to be passed back to the instruction immediately following the Call Internal instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Call Program with Variable Length Argument List (CALLPGMV)

Bound program access

```
Built-in number for CALLPGMV is 668.
CALLPGMV (
          pgmOrTmpltPtr : address of system pointer,
                          address of space pointer(16), or
                          address of open pointer
          argArray      : address of array of space pointer(16)
          nargs         : unsigned binary(4)
)
```

Description: Preserve the calling invocation and pass control to either the program entry procedure of a bound program or the external entry point of a non-bound program identified by *pgmOrTmpltPtr*.

The *pgmOrTmpltPtr* may address a system pointer which directly addresses the program that is to receive control, or may address a space pointer(16) to a call template which identifies the program to receive control. Specifying a template allows for additional controls over how the specified program is to be invoked. The format of the template is described under the **CALLX** instruction.

If the program to receive control is a Java(TM) program or a bound program that does not contain a program entry procedure, an *invalid operation for program* (hex 2C15) exception is signalled.

The number of arguments for the program call is given by *nargs*. The arguments are given by elements in *argArray*. The i'th argument of the OPM parameter list is constructed from the address of the i'th element of the array. Each element of the array is a space pointer(16) value.

The actual number of arguments passed, *nargs*, must be less than or equal to 16383 or else an *argument list length modification violation* (hex 0803) exception is signalled.

An *unsupported space use* (hex 0607) exception is signalled if this call would pass a parameter stored in teraspace to a program which is not teraspace capable. To be teraspace capable, a non-bound program must be created as teraspace capable or a bound program must be created with a teraspace capable program entry procedure.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute
  –
  – Program referenced by *pgmOrTmpltPtr*
  – Context referenced for address resolution

Lock Enforcement

•

• Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0802 Argument List Length Violation

0803 Argument List Length Modification Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2A Program Creation

2AB5 Observable Information Necessary For Retranslation Not Encapsulated

2C Program Execution

2C15 Invalid Operation for Program

2C1D Automatic Storage Overflow

2C1E Activation Access Violation

2C1F Program Signature Violation

2C20 Static Storage Overflow

2C21 Program Import Invalid

2C22 Data Reference Invalid

2C23 Imported Object Invalid

2C24 Activation Group Export Conflict

2C25 Import Not Found

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Check Lock Value (CHKLKVAL)

Bound program access
Built-in number for CHKLKVAL is 677. CHKLKVAL (addr : address of a signed binary(8) value (has alignment restrictions - see description below) old_val : signed binary(8) value new_val : signed binary(8) value) : signed binary(4)

Description: Performs the following atomic (uninterruptible) sequence of operations: The value pointed to by *addr* is compared to the *old_val* value. If the two values are equal, the *new_val* value is stored into the *addr* location and the numeric value 0 is returned. If the two values are not equal, the numeric value 1 is returned.

The first operand must be 8-byte aligned. Failure to have the first operand aligned properly will not be detected, but the results of the instruction are undefined when this occurs.

The comparison and conditional update of the first operand are performed atomically (not interruptible). This is important when multiple threads share the storage pointed to by *addr*. See "Atomicity" on page 1275 for additional information.

This operation is storage synchronizing. Any shared storage reads performed after a successful update of the lock value will be no less current than the most recent synchronizing action by the writer of the shared storage.

The behavior of this instruction is similar to the CMPSW instruction. CHKLKVAL is designed specifically for implementation of low-level locking protocols, and may perform better than using CMPSW for that purpose.

For correct storage synchronization, the CHKLKVAL instruction is commonly used in conjunction with the CLRLKVAL instruction.

See Storage Synchronization Concepts for additional information on storage synchronization.

Warning: Temporary Level 3 Header

Usage Notes

The CHKLKVAL and CLRLKVAL instructions are designed primarily to be used in combination when implementing low-level locking protocols to protect space data shared by two or more threads.

A typical usage pattern for these instructions is:

```
// Acquire the lock
loop until CHKLKVAL(LOCK, 0, 1) returns the value zero

[ Shared data reads/writes go here ]

// Release the "lock"
CLRLKVAL(LOCK, 0)
```

Where:

```
LOCK        is the address of an 8-byte variable shared by threads
              that want to enforce mutually exclusive access to a
              shared data structure.
```

Note that the example above is only a framework that illustrates a simple locking protocol. When all threads which share a data structure use this pattern, access to the data structure will be synchronized and free of race conditions.

The values 0 and 1 used above do not have any particular meaning to the instruction. They are simply unique values that are used in this example to represent an unlocked and locked state, respectively.

Note also that the pattern above contains no provision for deadlock detection/prevention, as would be available with higher level MI locking mechanisms, such as the LOCKMTX and LOCKSL instructions.

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2C Program Execution

- 2C04 Branch Target Invalid

36 Space Management

- 3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Cipher (CIPHER)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10EF	Receiver	Controls	Source

Operand 1: Space pointer data object.

Operand 2: Character(32, 42, 96) variable scalar.

Operand 3: Space pointer data object.

Bound program access

```
Built-in number for CIPHER is 176.
CIPHER (
        receiver   : address of space pointer(16)
        controls   : address
        source     : address of space pointer(16)
)
```

Description: The cipher operation specified in the *controls* (operand 2) is performed on the string value addressed by the *source* (operand 3). The result is placed into the string addressed by the *receiver* (operand 1).

The *controls* operand must be a character variable scalar. It specifies information to be used to control the cipher operation. The common header of the *controls* operand has the following format.

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Function identifier The *function identifier* must be hex 0002, hex 0005, hex 0007, hex 0008, hex 0010, hex 0011, hex 0013, or hex 0015. If not, a *template value invalid* (hex 3801) exception is signaled.	Char(2)
2	2	—- End —-	

The **function identifier** specifies the cryptographic service provider (CSP) for the cipher operation. It must specify hex 0002, hex 0005, hex 0007, hex 0008, hex 0010, hex 0011, hex 0013, or hex 0015. Any other value causes a *template value invalid* (hex 3801) exception to be signaled.

Hex 0002	The Machine CSP licensed internal code is to be used for a one-way encryption operation using the ANSI (American National Standards Institute) DEA (Data Encryption Algorithm).
Hex 0005	The Machine CSP licensed internal code is to be used to perform a one-way hash operation. The returned output may be a hash value or an HMAC (Hash Message Authentication Code) value. The supported hash algorithms are MD5 (Message Digest) and SHA-1 (Secure Hash Algorithm).
Hex 0007	The Machine CSP licensed internal code is to be used to perform a UNIX(R) crypt(3) operation.

Hex 0008	The Machine CSP licensed internal code is to be used to perform a pseudorandom number generator operation.
Hex 0010	The Machine CSP licensed internal code is to be used for an encryption or decryption operation using the ANSI (American National Standards Institute) DEA (Data Encryption Algorithm). In order to use this function identifier, the cryptography attributes must contain an algorithm entry that specifies DES (Data Encryption Standard) is provided by the machine service provider. If no such entry exists, then *requested function not valid* (hex 1C08) exception will be signaled. The cryptography attributes may be materialized through the use of the Materialize Machine Attributes (MATMATR) instruction and using a selection value of hex 01C8.
Hex 0011	The Machine CSP licensed internal code is to be used for an encryption or decryption operation using the ANSI (American National Standards Institute) TDEA (Triple Data Encryption Algorithm). In order to use this function identifier, the cryptography attributes must contain an algorithm entry that specifies TDES is provided by the machine service provider. If no such entry exists, then *requested function not valid* (hex 1C08) exception will be signaled. The cryptography attributes may be materialized through the use of the Materialize Machine Attributes (MATMATR) instruction and using a selection value of hex 01C8.
Hex 0013	The Machine CSP licensed internal code is to be used for an encryption or decryption operation using an algorithm compatible with RC4(R). In order to use this function identifier, the cryptography attributes must contain an algorithm entry that specifies the RC4-compatible algorithm is provided by the machine service provider. If no such entry exists, then *requested function not valid* (hex 1C08) exception will be signaled. The cryptography attributes may be materialized through the use of the Materialize Machine Attributes (MATMATR) instruction and using a selection value of hex 01C8.
Hex 0015	The Machine CSP licensed internal code is to be used for an encryption or decryption operation using the Advanced Encryption Standard (AES). In order to use this function identifier, the cryptography attributes must contain an algorithm entry that specifies AES is provided by the machine service provider. If no such entry exists, then *requested function not valid* (hex 1C08) exception will be signaled. The cryptography attributes may be materialized through the use of the Materialize Machine Attributes (MATMATR) instruction and using a selection value of hex 01C8.

The format of the *controls* operand is dependent on the value of the *function identifier*.

Warning: Temporary Level 3 Header

Function Identifier 0002

The following description applies only to function identifier 0002.

The *controls* operand must have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(32)	
0	0		Function identifier	Char(2)
2	2		Data length	Char(2)
4	4		Options	Char(1)
4	4		Reserved (binary 0)	Bit 0
4	4		Use cipher block chaining	Bit 1

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = No **1 =** Yes	
4	4		Data padding	Bit 2
			0 = No **1 =** Yes	
4	4		Reserved (binary 0)	Bits 3-7
5	5		Cryptographic key	Char(8)
13	D		Reserved (ignored)	Char(1)
14	E		Initial chaining value	Char(8)
22	16		Pad character	Char(1)
23	17		Reserved (binary 0)	Char(9)
32	20	—- End —-		

The first character of the *source* and *receiver* strings is addressed by their respective operand pointers. The **data length** field of the *controls* operand specifies the length of the input *source* data. The length of data returned in the *receiver* is determined from the length of the *source*. When the *data padding* field specifies *no*, the length of data returned in the *receiver* is equal to the length of the *source*. When the *data padding* field specifies *yes*, the length of data returned in the *receiver* is not equal to the length of the *source* and is returned in the *data length* field of the *controls* operand. Refer to the discussion of the *data padding* field for details on the amount of data returned in this case.

The *data length* field specifies the length of the data addressed by the *source* operand. The *data length* value must be nonzero and less than or equal to 64 bytes. In addition, when the *data padding* and *use cipher block chaining* fields specify *no*, the *data length* must be a multiple of 8 bytes. An incorrect *data length* value results in the signaling of the *template value invalid* (hex 3801) exception. When the *data padding* field specifies *yes*, the length of the data placed into the *receiver* is returned in this field.

The **use cipher block chaining** field specifies whether or not cipher block chaining is to be used during the cipher operation.

When the *use cipher block chaining* field specifies *yes*, the first block of data from the *source* operand is exclusive ORed with the *initial chaining value* and then encrypted. For subsequent blocks of data, the prior block of encrypted data from the *receiver* operand is exclusive ORed with the current data block from the *source* operand and the result is encrypted.

The **data padding** field specifies whether data padding is to be used during the cipher operation. When the *data padding* field specifies *no*, padding is not performed. When the *data padding* field specifies *yes*, padding is performed. In this case, the length of data returned in the receiver is different from the source length and is returned in the *data length* field for both encrypt and decrypt operations.

When the *data padding* field specifies *yes*, the data from the *source* operand is padded out to the next multiple of 8 bytes; for example, a source length of 20 is padded to 24, 32 is padded to 40, and so forth. The final block of *source* data is padded with zero to seven repetitions of the *pad character* until the block length is 7 bytes in length. The eighth byte is then filled with a 1-byte binary counter containing the number of pad characters used (a value from one to eight which includes the 1-byte counter) and the block is encrypted.

The **cryptographic key** field specifies the key to be used for the cipher operation. The *cryptographic key* is provided in an unencrypted form.

The **initial chaining value** field specifies the 8-byte value to be used in conjunction with cipher block chaining when the *use cipher block chaining* field specifies *yes*. When the *use cipher block chaining* field specifies *no*, this field is ignored. Refer to the description of the *use cipher block chaining* field for details on how this value is used in the cipher operation.

The **pad character** field specifies the value to be used as a pad character when the *data padding* field specifies *yes*. When the *data padding* field specifies *no*, this field is ignored.

Specific Properties of ANSI DEA: The encrypt operation is performed iteratively upon 8-byte blocks of the *source* operand. Each block is encrypted using DEA and the information specified in the *controls* and the resulting value is placed into the *receiver* at the same relative location as that from which the *source* data was accessed from the *source* operand. The process is repeated until the data in the *source* is exhausted.

The key is presented to the DEA as a 64-bit value. The DEA uses the first 7 bits of each byte, for a total of 56 bits, as the key. The remaining 8 bits enforce odd parity of each byte when required. The DEA uses the key and the input data to calculate the output. Given fixed input data, the output is unique for each unique set of 56 bits.

Refer to *Cryptographic Support/400 User's Guide* (SC41-3342) for more information on the DEA.

Function Identifier 0005

The following description applies only to function identifier 0005.

The *controls* operand must be 16-byte aligned and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(96)	
0	0		Function identifier	Char(2)
2	2		Hash algorithm	Char(1)
			Hex 00 = MD5	
			Hex 01 = SHA-1	
3	3		Sequence	Char(1)
			Hex 00 = Only	
			Hex 01 = First	
			Hex 02 = Middle	
			Hex 03 = Final	
4	4		Data length	UBin(4)
8	8		Output	Char(1)
			Hex 00 = Hash	
			Hex 01 = HMAC	
9	9		Reserved (ignored)	Char(7)
16	10		Hash context	Space pointer
32	20		HMAC key	Space pointer
48	30		HMAC key length	UBin(4)

Offset Dec	Hex	Field Name	Data Type and Length
96	60	—- End —-	

The **hash algorithm** field specifies the one-way hash function to perform. A hash function takes a variable-length input string and converts it to a fixed-length output string. A one-way hash function means the function is for all practical purposes irreversible in that it is computationally infeasible to re-create the input message from the hash value or to find another message that will hash to the same value.

MD5 produces a 128-bit hash value. MD5 is documented in RFC 1321.

SHA-1 produces a 160-bit hash value. SHA-1 is documented in FIPS 180-1.

A hash of data may be performed in one execution of the CIPHER instruction or in several which allows the hash of data that does not lie in contiguous storage. This is specified using the **sequence** field. When performing the hash in one execution of CIPHER, the *sequence* field should specify *only*. Otherwise, the first use of the CIPHER instruction should specify *first*, the last use of CIPHER should specify *final*, and any executions of CIPHER in between should specify *middle*. The hash will be returned in the *receiver* operand when the *sequence* field specifies *only* or *final*.

The **data length** field specifies the length of the input *source* data.

The **output** field specifies the value to return in the *receiver* operand. When *hash* is specified, the hash of the *source* string is returned in the *receiver*. When *HMAC* is specified, the HMAC of the *source* string is returned in the *receiver*. HMAC is a mechanism for message authentication using a one-way hash function and a secret shared key. It is documented in RFC 2104. For both *hash* and *HMAC*, if *MD5* is specified for the *hash algorithm*, 16 bytes are returned in the *receiver*. If *SHA-1* is specified, 20 bytes are returned in the *receiver*.

The **hash context** space pointer points to a work area belonging to the user. If the *output* field specifies *hash* this work area must be at least 96 bytes long. If *HMAC* is specified, it must be at least 160 bytes long. Prior to executing CIPHER with *only* or *first* specified in the *sequence* field, the work area should be set to binary 0s. When executing CIPHER with *middle* or *final* specified in the *sequence* field, the *hash context* field should point to the work area that was used on the previous execution of CIPHER. The user should not modify data returned in the work area, or unpredictable results may occur.

The **HMAC key** space pointer points to an area containing the secret key to be used in an HMAC operation. This field is ignored when the *output* field specifies *hash* or if the *sequence* field specifies *middle* or *final*.

The **HMAC key length** field specifies the length of the *HMAC key*. It is ignored when the *output* field specifies *hash* or if the *sequence* field specifies *middle* or *final*. The minimum size is 16 bytes when using the *MD5 hash algorithm*, and 20 bytes when using the *SHA-1 hash algorithm*. An incorrect *HMAC key length* value results in the signaling of the *template value invalid* (hex 3801) exception. Keys longer than these sizes do not significantly increase the function strength unless the randomness of the key is considered weak. In accordance with the RFC, a key longer than 64 bytes will be hashed before it is used.

Function Identifier 0007

The following description applies only to function identifier 0007.

The *controls* operand must be 16-byte aligned and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(32)	
0	0		Function identifier	Char(2)
2	2		Reserved (binary 0)	Char(2)
4	4		Salt	Char(2)
6	6		Reserved (binary 0)	Char(26)
32	20	—- End —-		

The crypt(3) function is a string encryption function used on UNIX(R) systems for password authentication. Crypt(3) is a one-way (no decryption) variant of DES (Data Encryption Standard).

Crypt(3) encrypts 8 bytes of hex 00 25 times using the 8-byte password pointed to by the *source* operand as the key for the DES algorithm. The password may be any value. If the password is under 8 bytes, the *source* operand should be padded on the right to 8 bytes with hex 00.

The **salt** value is used to modify the DES E bit-selection table in one of 4096 possible ways. Each *salt* byte must be an ASCII character, "a"-"z", "A"-"Z", "0"-"9", "." or "/". An invalid *salt* value will produce a *template value invalid* (hex 3801) exception. For a description of the DES (Data Encryption Standard) algorithm, including the E bit-selection table, see the Federal Information Processing Standard (FIPS) 46-2.

The result of the encryption operation is converted into 11 bytes of ASCII characters. At the completion of the crypt(3) operation, the *receiver* operand will contain the salt value followed by the resultant ASCII characters, for a total of 13 bytes.

Function Identifier 0008

The following description applies only to function identifier 0008.

The *controls* operand must be 16-byte aligned and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(32)	
0	0		Function identifier	Char(2)
2	2		Seed request	Char(1)
			Hex 00 = No seed	
			Hex 01 = Add seed	
3	3		Reserved (binary 0)	Char(3)
6	6		Seed length	UBin(2)
8	8		PRN (pseudorandom number) request	Char(1)
			Hex 00 = Generate real pseudorandom numbers	
			Hex 01 = Generate test pseudorandom numbers	
9	9		PRN parity	Char(1)

Offset Dec	Hex	Field Name	Data Type and Length	
			Hex 00 = No parity	
			Hex 01 = Odd parity	
			Hex 02 = Even parity	
10	A		Reserved (binary 0)	Char(4)
14	E		Number of PRNs	UBin(2)
16	10		Reserved (binary 0)	Char(16)
32	20	—- End —-		

The Pseudorandom Number Generator is composed of two parts - pseudorandom number generation and seed management. Pseudorandom number generation is performed using the FIPS 186-1 algorithm. Cryptographically secure pseudorandom numbers rely on good seed. The FIPS 186-1 key and seed values are obtained from the system seed digest. The system automatically generates the system seed digest using data collected from system information, or by using the random number generator on a cryptographic coprocessor if one is available. System-generated seed can never be truly random and if a cryptographic coprocessor is not available, a user may use this interface to add their own random seed to the system seed digest. This should be done as soon as possible anytime the system seed digest is created. The system seed digest is created during the first IPL after an install of the Licensed Internal Code, or if ever destroyed.

The **seed request** field indicates if user seed is being added to the system seed digest. A *no seed* value indicates no seed data is being added on this request. An *add seed* value indicates seed data is being added. The seed data is obtained from the *source* operand. Any other values will produce a *template value invalid* (hex 3801) exception. All object authority special authority is required for an *add seed* request. If *add seed* is specified and the issuer does not have all object special authority, a *special authorization required* (hex 0A04) exception is signalled.

The **seed length** field indicates the number of seed data bytes in the *source* operand. This field is ignored for the *no seed* option. If the *add seed* option is specified and the *seed length* is zero, no seed is added.

It is important that the seed data be unpredictable and have as much entropy as possible. Entropy is the minimum number of bits needed to represent the information contained in some data. For the purpose of this instruction, entropy is a measure of the amount of uncertainty or unpredictability of the seed. The system seed digest holds a maximum of 320 bits of entropy. To totally refresh the system seed digest, you should add at least that much entropy. Possible sources of seed data are coin flipping, keystroke or mouse timings, or a noise source such as on the 4758 cryptographic coprocessor.

The **PRN (pseudorandom numbers) request** field is used to request output of PRNs. A *generate real pseudorandom numbers* value indicates real pseudorandom numbers should be output. A *generate test pseudorandom numbers* value indicates test pseudorandom numbers should be output. Test pseudorandom numbers are produced using fixed FIPS 186-1 key and seed values. The test pseudorandom numbers will be statistically random. However, the next request for test pseudorandom numbers will return an identical stream of pseudorandom numbers. Any other values specified for the *PRN request* field will signal a *template value invalid* (hex 3801) exception. This field is ignored if the *number of PRNs* field is 0.

The *PRN request* is performed after the *seed request* has completed.

*PRN request*s for *generate real pseudorandom numbers* will result in a *requested function not valid* (hex 1C08) exception if the system seed digest is not fully initialized.

The **number of PRNs** field indicates the number of pseudorandom number bytes to return. If 0 is specified, no pseudorandom numbers are produced. Pseudorandom numbers are returned in the *receiver* operand.

The **PRN parity** field indicates how the parity of the pseudorandom numbers should be set. If a *no parity* value is specified, the pseudorandom number output is not altered. If an *odd parity* value is specified, each byte will be set to odd parity by altering the low order bit as needed. If an *even parity* value is specified, each byte will be set to even parity by altering the low order bit as needed. Any other values will produce a *template value invalid* (hex 3801) exception.

Reserved fields must be set to binary 0 or a *template value invalid* (hex 3801) exception will be signalled.

Function Identifier 0010

The following description applies only to function identifier 0010.

The *controls* operand must have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(42)	
0	0		Function identifier	Char(2)
2	2		Data length	UBin(2)
4	4		Operation	Char(1)
			Hex 00 = Encrypt	
			Hex 01 = Decrypt	
			Hex 02 = MAC (Message Authentication Code)	
5	5		Mode	Char(1)
			Hex 00 = ECB (Electronic Codebook)	
			Hex 01 = CBC (Cipher Block Chaining)	
			Hex 02 = OFB (Output Feedback)	
			Hex 03 = CFB (Cipher Feedback) 1-bit	
			Hex 04 = CFB (Cipher Feedback) 8-bit	
			Hex 05 = CFB (Cipher Feedback) 64-bit	
6	6		Initialization vector	Char(8)
14	E		Pad option	Char(1)
			Hex 00 = No pad	
			Hex 01 = Pad using pad character	
			Hex 02 = Pad using pad number	
15	F		Pad character	Char(1)
16	10		MAC (Message Authentication Code) length	Char(1)
17	11		Reserved (binary 0)	Char(1)

Offset Dec	Hex	Field Name	Data Type and Length	
18	12		Key	Char(8)
26	1A		Reserved (binary 0)	Char(16)
42	2A	—- End —-		

Encryption, decryption, or creation of a message authentication code (MAC) is performed as specified in the **operation** field using the DES algorithm as defined in FIPS PUB 46-3 Data Encryption Standard and in ANSI X3.92 Data Encryption Algorithm (DEA). DES must be enabled in the cryptographic attributes for an *encrypt* or *decrypt* operation or else a *requested function not valid* (hex 1C08) exception is signaled. To query the cryptographic attributes, use the MATMATR instruction with a selection value of hex 01C8. A *MAC* operation is always allowed regardless of the cryptographic attributes.

The **data length** field specifies the length of the input data pointed to by the *source* operand. Upon completion of the operation, the *data length* field will be set with the length of data returned in the *receiver* operand. When *CFB 1-bit* is specified for the *mode* field, the *data length* field is specified in bits, otherwise it is specified in bytes. When the *mode* is *ECB*, *OFB*, or *CFB 64-bit* and the *pad option* is *no pad*, the value of the *data length* field must be a multiple of 8, otherwise a *template value invalid* (hex 3801) exception will be signaled.

The **mode** field specifies the mode of operation as defined in FIPS PUB 81 (also ANSI X3.106). Valid values are *ECB* for Electronic Codebook; *CBC* for Cipher Block Chaining; *OFB* for Output Feedback; and *CFB 1-bit*, *CFB 8-bit*, and *CFB 64-bit* for Cipher Feedback. Refer to the standard for an explanation of these modes. *ECB* and *OFB* are not valid when the *operation* field specifies *MAC*.

For all *mode* values except *ECB*, the **initialization vector** (IV) will be used as part of the operation. Refer to FIPS PUB 81 for an explanation of its use. The IV need not be secret, but it should be unique. If not unique, it may compromise security. The IV can be any binary value. Upon completion of the operation, an output chaining value will be returned in the *initialization vector* field. This value can be used as the IV for the next DES operation when encrypting, decrypting, or MACing a message in multiple blocks.

Specifying a **pad option** will pad the data in the *source* operand out to the next 8 byte multiple when encrypting or MACing. When decrypting, specifying a *pad option* will strip the pad bytes off the end of the output data before returning it in the *receiver* operand. For example, a source length of 20 is padded to 24, 32 is padded to 40, and so forth, when encrypting. The last byte of pad data is filled with a 1-byte binary counter containing the number of pad characters used (a value from 1 to 8 which includes the 1-byte counter). If *pad option* specifies *pad using pad character*, the **pad character** field is used for the preceeding pad characters. If the *pad option* specifies *pad using pad number*, the value of the last byte (the pad counter) is used for the preceeding pad characters. When decrypting, it is not necessary to know which pad method was used when the data was encrypted. If the data was padded, you can specify either *pad using pad character* or *pad using pad number*. The *pad option* is ignored when *CFB 1-bit* or *CFB 8-bit* is specified for *mode*.

When the *operation* field specifies *MAC*, the **MAC length** field specifies the length of the MAC to return in the *receiver operand*. Otherwise, the field is ignored. When MACing, the *source* operand data is encrypted in the normal manner. From the last 8 bytes of the encrypted data, the leftmost *MAC length* bytes are returned. Valid values for *MAC length* are 1 to 8.

The key for the DES operation is specified in the **key** field. The key can be any binary value. Note, only the leftmost 7 bits from each byte are used for the key. The rightmost bit of each byte is used to enforce parity when required.

To obtain good random key and IV values, use CIPHER function identifier hex 0008.

Reserved fields must be set to binary 0 or a *template value invalid* (hex 3801) exception will be signalled.

Function Identifier 0011

The following description applies only to function identifier 0011.

The *controls* operand must have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(42)	
0	0		Function identifier	Char(2)
2	2		Data length	UBin(2)
4	4		Operation	Char(1)
			Hex 00 = Encrypt	
			Hex 01 = Decrypt	
			Hex 02 = MAC (Message Authentication Code)	
5	5		Mode	Char(1)
			Hex 00 = ECB (Electronic Codebook)	
			Hex 01 = CBC (Cipher Block Chaining)	
			Hex 02 = OFB (Output Feedback)	
			Hex 03 = CFB (Cipher Feedback) 1-bit	
			Hex 04 = CFB (Cipher Feedback) 8-bit	
			Hex 05 = CFB (Cipher Feedback) 64-bit	
6	6		Initialization vector	Char(8)
14	E		Pad option	Char(1)
			Hex 00 = No pad	
			Hex 01 = Pad using pad character	
			Hex 02 = Pad using pad number	
15	F		Pad character	Char(1)
16	10		MAC (Message Authentication Code) length	Char(1)
17	11		Key option	Char(1)
			Hex 01 = One key	
			Hex 02 = Two keys	
			Hex 03 = Three keys	
18	12		Key 1	Char(8)
26	1A		Key 2	Char(8)
34	22		Key 3	Char(8)
42	2A	—- End —-		

Encryption, decryption, or the creation of a message authentication code (MAC) is performed as specified in the **operation** field using the Triple DES (TDES) algorithm as defined in FIPS PUB 46-3 Data Encryption Standard and in ANSI X9.52 Triple Data Encryption Algorithm Modes of Operation (TDEA).

Triple DES must be enabled in the cryptographic attributes for an *encrypt* or *decrypt operation* or else a *requested function not valid* (hex 1C08) exception is signaled. To query the cryptographic attributes, use the MATMATR instruction with a selection value of hex 01C8. A *MAC operation* is always allowed regardless of the cryptographic attributes.

The **data length** field specifies the length of the input data pointed to by the *source* operand. Upon completion of the operation, the *data length* field will be set with the length of data returned in the *receiver* operand. When *CFB 1-bit* is specified for the *mode* field, the *data length* field is specified in bits, otherwise it is specified in bytes. When the *mode* is *ECB*, *OFB*, or *CFB 64-bit* and the *pad option* is *no pad*, the value of the *data length* field must be a multiple of 8, otherwise a *template value invalid* (hex 3801) exception will be signaled.

The **mode** field specifies the mode of operation as defined in ANSI X9.52. Valid values are *ECB* for Electronic Codebook; *CBC* for Cipher Block Chaining; *OFB* for Output Feedback; and *CFB 1-bit*, *CFB 8-bit*, and *CFB 64-bit* for Cipher Feedback. Refer to the standard for an explanation of these modes. *CBC* must be specified when the *operation* field specifies *MAC*.

For all *mode* values except *ECB*, the **initialization vector** (IV) will be used as part of the operation. Refer to ANSI X9.52 for an explanation of its use. The IV need not be secret, but it should be unique. If not unique, it may compromise security. The IV can be any binary value. Upon completion of the operation, an output chaining value will be returned in the *initialization vector* field. This value can be used as the IV for the next TDES operation when encrypting, decrypting, or MACing a message in multiple blocks.

Specifying a **pad option** will pad the data in the *source* operand out to the next 8 byte multiple when encrypting. When decrypting, specifying a **pad option** will strip the pad bytes off the end of the output data before returning it in the *receiver* operand. For example, a source length of 20 is padded to 24, 32 is padded to 40, and so forth, when encrypting. The last byte of pad data is filled with a 1-byte binary counter containing the number of pad characters used (a value from 1 to 8 which includes the 1-byte counter). If *pad option* specifies *pad using pad character*, the **pad character** field is used for the preceeding pad characters. If the *pad option* specifies *pad using pad number*, the value of the last byte (the pad counter) is used for the preceeding pad characters. When decrypting, it is not necessary to know which pad method was used when the data was encrypted. If the data was padded, you can specify either *pad using pad character* or *pad using pad number*. The *pad option* is ignored when *CFB 1-bit* or *CFB 8-bit* is specified for *mode*. If a *MAC operation* is specified, *pad option* is ignored. If the *data length* is not a multiple of 8, the data will be padded with hex 00s.

When the *operation* field specifies *MAC*, the **MAC length** field specifies the length of the MAC to return in the *receiver operand*. Otherwise, the field is ignored. When MACing, the *source* operand data minus the last 8-byte block is encrypted using DES. The last block is encrypted using TDES. From the last resulting block, the leftmost *MAC length* bytes are returned. Valid values for *MAC length* are 1 to 8.

The Triple DES key is specified in the **key 1**, **key 2**, and **key 3** fields. If **key option** specifies *three keys*, all three key fields are used for the key. If *key option* specifies *two keys*, *key 1* will be used for *key 3*. If *key option* specifies *one key*, *key 1* will be used for *key 2* and *key 3*. (This last option is equivalent to performing a single DES operation.) Note, only the leftmost 7 bits from each byte are used for the key. The rightmost bit of each byte is used to enforce parity when required.

To obtain good random key and IV values, use CIPHER function identifier hex 0008.

Reserved fields must be set to binary 0 or a *template value invalid* (hex 3801) exception will be signalled.

Function Identifier 0013

The following description applies only to function identifier 0013.

The *controls* operand must be 16-byte aligned and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(32)	
0	0		Function identifier	Char(2)
2	2		Data length	UBin(2)
4	4		Operation	Char(1)
			Hex 00 = Encrypt	
			Hex 01 = Decrypt	
5	5		Reserved (binary 0)	Char(11)
16	10		Key context pointer	Space pointer
32	20	—- End —-		

Encryption or decryption is performed as specified in the **operation** field using an RC4(R)-compatible algorithm.

RC4 must be enabled in the cryptographic attributes or else a *requested function not valid* (hex 1C08) exception is signaled. To query the cryptographic attributes, use the MATMATR instruction with a selection value of hex 01C8.

The **data length** field specifies the length of the input data pointed to by the *source* operand. The encrypted or decrypted data is returned in the area pointed to by the *receiver* operand and is identical in length. If data length is 0, no data is encrypted or decrypted, but the *key context* will be initialized.

The **key context pointer** field points to a 264-byte area belonging to the user and having the following format.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Key context	Char(264)	
0	0		Key stream	Char(256)
256	100		Key length	UBin(2)
258	102		Reserved (binary 0)	Char(6)
264	108	—- End —-		

The **key context** allows encryption or decryption of a message in multiple blocks using multiple calls to CIPHER. Prior to the first call to CIPHER, set the key in the **key stream** field. The key may be any binary value and any length from 1 to 16 bytes depending on the length enabled in the cryptographic attributes. To query the cryptographic attributes, use the MATMATR instruction with a selection value of hex 01C8. Specify the length of the supplied key in the **key length** field. To obtain good random key values, use CIPHER function identifier hex 0008.

The *key context* contains the state of the encryption or decryption operation. As data is encrypted or decrypted, the *key stream* is altered by the RC4-compatible algorithm. Consequently, when encrypting or decrypting a message in multiple blocks, subsequent calls to CIPHER must pass in the identical *key context* returned from the previous call.

Because of the nature of the RC4-compatible algorithm, using the same key for more than one message will severely compromise security.

Reserved fields of the *key context* must be set to binary 0 or a *template value invalid* (hex 3801) exception will be signaled.

Function Identifier 0015

The following description applies only to function identifier 0015.

The *controls* operand must be 16-byte aligned and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(96)	
0	0		Function identifier	Char(2)
2	2		Data length	UBin(2)
4	4		Operation	Char(1)
			Hex 00 = Encrypt	
			Hex 01 = Decrypt	
			Hex 02 = MAC (Message Authentication Code)	
5	5		Mode	Char(1)
			Hex 00 = ECB (Electronic Codebook)	
			Hex 01 = CBC (Cipher Block Chaining)	
6	6		Block length	Char(1)
7	7		MAC (Mesage Authentication Code) length	Char(1)
8	8		Initialization vector	Char(32)
40	28		Reserved (binary 0)	Char(7)
47	2F		Key option	Char(1)
			Hex 00 = Use Key schedule	
			Hex 10 = Use 16-byte key	
			Hex 18 = Use 24-byte key	
			Hex 20 = Use 32-byte key	
48	30		Key schedule	Space pointer
64	40		Key	Char(32)
96	60	—- End —-		

Encryption, decryption, or the creation of a message authentication code (MAC) is performed as specified in the **operation** field using the NIST-proposed Advanced Encryption Standard (AES) algorithm.

AES must be enabled in the cryptographic attributes for an *encrypt* or *decrypt operation* or else a *requested function not valid* (hex 1C08) exception is signaled. To query the cryptographic attributes, use the MATMATR instruction with a selection value of hex 01C8. A *MAC operation* is always allowed regardless of the cryptographic attributes.

The **data length** field specifies the length of the input data pointed to by the *source* operand. The data length can be 0 or a multiple of the *block length*, otherwise a *template value invalid* (hex 3801) exception will be signaled. If 0, no data is encrypted or decrypted, but the *key schedule* will be calculated.

The **mode** field specifies the mode of operation as defined in FIPS PUB 81. Valid values are *ECB* for Electronic Codebook; and *CBC* for Cipher Block Chaining. Refer to the standard for an explanation of these modes. *CBC mode* must be specified when *operation* is *MAC*.

When *CBC* is specified, the **initialization vector** (IV) will be used as part of the operation. The length of IV used is that specified in *block length*. Refer to ANSI X9.52 for an explanation of its use. The IV need not be secret, but it should be unique. If not unique, it may compromise security. The IV can be any binary value. Upon completion of an AES CBC operation, an output chaining value will be returned in the *initialization vector* field. This value should be used as the IV for the next AES operation when encrypting, decrypting, or MACing a message in multiple blocks.

Block length indicates the number of bytes that are encrypted/decrypted at one time. Supported block lengths are 16, 24, and 32 (hex 10, 18, and 20). Other lengths will produce a *template value invalid* (hex 3801) exception.

When the *operation* specifies *MAC*, the **MAC length** field specifies the length of MAC to return in the *receiver* operand. Otherwise, the field is ignored. When MACing, the *source* operand data is encrypted in the normal manner. From the last *block length* bytes of encrypted data, the leftmost *MAC length* bytes are returned. Valid values for *MAC length* are 1 to *block length*. Other values will cause a *template value invalid* (hex 3801) exception.

The AES key is specified in the **key** field and should be left justified. The length of key may be 16, 24, or 32 bytes as specified in the **key option** field. The initial step in an AES operation is to calculate a set of subkeys. The subkeys will be stored in the area pointed to by the **key schedule** pointer, if the pointer value is not a null pointer value. This area should be 4-byte aligned and 1088 bytes in length. This allows subsequent AES operations to specify the *use key schedule* option and bypass the subkey generation step. A *template value invalid* (hex 3801) exception will be signalled if *use key schedule* is specified but the *key schedule* pointer is null.

To obtain good random key and IV values, use CIPHER function identifier hex 0008.

Reserved fields must be set to binary 0 or a *template value invalid* (hex 3801) exception will be signalled.

Limitations (Subject to Change)

The following are limits that apply to the functions performed by this instruction.

Valid results are produced for the case of the receiver and source operands being coincident with one another. The source data is accessed first, then the result is stored in the receiver.

Partial overlap between the source and receiver operands may produce invalid results.

Authorization Required

•

- All Object Special Authority
  –
  – *Add seed* specified in the *seed request* of the *controls* operand for function identifier Hex 0008.

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

- 0A01 Unauthorized for Operation
- 0A04 Special Authorization Required

0C Computation

- 0C0F Master Key Not Defined

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1A Lock State

- 1A01 Invalid Lock State

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded
- 1C08 Requested Function Not Valid

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid
- 2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

2E02 Security Audit Journal Failure

32 Scalar Specification

3201 Scalar Type Invalid

34 Source/Sink Management

3403 Source/Sink Object State Invalid

3404 Source/Sink Resource Not Available

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Clear Bit in String (CLRBTS)

Op Code (Hex)	Operand 1	Operand 2
102E	Receiver	Offset

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Binary scalar.

Bound program access

```
Built-in number for CLRBTS is 2.
CLRBTS (
        receiver   : address
        offset     : unsigned binary(4)
)
```

The *offset* parameter must be between 0 and 65,535.

Description: Clears the bit of the *receiver* operand as indicated by the bit *offset* operand.

The selected bit from the *receiver* operand is set to a value of binary 0.

The *receiver* operand can be character or numeric. The leftmost bytes of the *receiver* operand are used in the operation. The *receiver* operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus 1.

The *receiver* cannot be a variable substring.

The *offset* operand indicates which bit of the *receiver* operand is to be cleared, with an offset of zero indicating the leftmost bit of the leftmost byte of the *receiver* operand.

If an offset value less than zero or beyond the length of the string is specified, a *scalar value invalid* (hex 3203) exception is signaled.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Clear Invocation Exit (CLRIEXIT)

Op Code (Hex)
0250

Description: The instruction removes the invocation exit program for the requesting invocation. No exception is signaled if an invocation exit program is not specified for the current invocation. Also, an implicit clear of the invocation exit occurs when the invocation exit program is given control, or the program which set the invocation exit completes execution.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2208 Object Compressed

220B Object Not Available

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

Clear Invocation Flags (CLRINVF)

Bound program access

`Built-in number for CLRINVF is 4.`
`CLRINVF (`
`  clear_mask  : unsigned binary(4) value which specifies the`
`  invocation flags to be cleared`
`)`

Description: Operand 1 selects which invocation flags are to be cleared. The invocation flags to be cleared are indicated with 0 values in the bit positions of operand 1, based on the correspondence described below. Only the invocation flags that are ″writeable″ can be cleared. Any ″read-only″ flags selected by the stack value are unchanged.

The operation is performed by doing a bit-wise Boolean **and** of the 16 writeable status bits with the low-order two bytes of the *clear mask* operand, and then replacing the writeable status bits with the result of this **and**.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

Clear Lock Value (CLRLKVAL)

Bound program access

```
Built-in number for CLRLKVAL is 678.
CLRLKVAL (
           addr    : address of a signed binary(8) value  (has
                       alignment restrictions - see description below)
           new_val : signed binary(8) value
)
```

Description: The value pointed to by *addr* is set to the *new_val* value.

The first operand must be 8-byte aligned. Failure to have the first operand aligned properly will not be detected, but the results of the instruction are undefined when this occurs.

This operation is storage synchronizing. When a thread performs the CLRLKVAL instruction, shared space data values previously written by that thread will be visible to other threads at their next synchronizing actions. The shared space data values seen by those other threads will be at least as current as what was written by the thread performing the CLRLKVAL.

The CLRLKVAL instruction is designed specifically for implementation of low-level locking protocols. For correct storage synchronization, the CLRLKVAL instruction is commonly used in conjunction with the CHKLKVAL instruction.

See Storage Synchronization Concepts for additional information on storage synchronization.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compare and Swap (CMPSW)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4 [5]
CMPSWB 1C37	Branch options	Compare operand 1	Compare operand 2	Swap operand	Branch target
CMPSWI 1837	Indicator options	Compare operand 1	Compare operand 2	Swap operand	Indicator target

Operand 1: Character(1,2,4,8) variable scalar.

Operand 2: Character(1,2,4,8) variable scalar.

Operand 3: Character(1,2,4,8) scalar.

Operand 4 [4-5]:

•

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CMPSWP is 156.
CMPSWP (
        op1   : address of a scalar(1,2,4,8) value
        op2   : address of a scalar(1,2,4,8) value (has alignment
                restrictions based on the length of the scalar - see
                description below)
        op3   : scalar(1,2,4,8) value
        cntl  : signed binary(4) literal value   (this operand
                is optional -- see description below)
) :   signed binary(4)
```

If the values of op1 and op2 are equal, the value 1 is returned. Otherwise, the value 0 is returned.

Description: The value of the first compare operand is compared with the value of the second compare operand. If they are equal, the *swap* operand is stored in the second compare operand's location. If they are unequal, the second compare operand is stored into the first compare operand's location. Based on the comparison, the resulting condition is used with the extender field to:

•

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

When an equal comparison occurs, it is assured that no access by another Compare and Swap instruction will occur at the second compare operand location between the moment that the second compare operand is fetched for comparison and the moment that the *swap* operand is stored at the second compare operand location.

When an unequal comparison occurs, no atomicity guarantees are made regarding the store to the first compare operand location and other Compare and Swap instruction access. Thus only the second compare operand should be a variable shared for concurrent processing control.

Both compare operands must be 1, 2, 4 or 8 byte character variable scalars and the *swap* operand must be a 1, 2, 4 or 8 byte character scalar. All three operands must have the same length. Failure to have the operands the same length will not be detected and the results of the Compare and Swap instruction are undefined when this occurs. The second operand must be aligned based on its length:

•

- one byte length - no alignment restrictions

- two byte length - halfword aligned
- four byte length - fullword aligned
- eight byte length - doubleword aligned

Failure to have the second operand aligned properly will not be detected, but the results of the Compare and Swap instruction are undefined when this occurs.

For bound program access, the *cntl* operand is optional, and does not need to be specified. If the *cntl* operand is specified, it has the following effect depending on the value:

•
- Hex 0x0 = (Default) Storage synchronization is performed both before and after a successful store of the *swap* operand.
- Hex 0x1 = No storage synchronization is performed

If the *cntl* operand is not specified, the instruction proceeds as if the default value had been specified.

The machine does not enforce that only the values enumerated above are used for the *cntl* operand. When specified, use of any value other than those enumerated above may result in unpredictable behavior.

Warning: Temporary Level 3 Header

Storage Synchronization

Unless explicitly disabled with the *no storage synchronization is performed* control option (disabling is possible only using bound program access), this instruction synchronizes storage both before and after the *swap* operand is stored to *op2*. Synchronized operation provides the following guarantees:

•
- When reading shared storage after a successful store of the *swap* operand, the thread performing the CMPSW will have a view of shared storage no less current then the most recent synchronizing actions taken by threads writing the shared storage.
- Shared data written by the thread performing the CMPSW prior to a successful store of the *swap* operand will be current from the perspective of other threads at their next synchronizing action.

When synchronization is disabled for CMPSW, storage synchronization can be accomplished using other storage synchronizing MI instructions, such as SYNCSTG.

See Storage Synchronization Concepts for additional information on storage synchronization.

Usage Note

Note that the compare and swap construct (including both this CMPSW instruction and support found on other architectures) simply compares values; it cannot determine whether the presence of the same value indicates that the data object for which an update is being synchronized is in fact the intended data object.

For example, CMPSW is insufficient for modifications to data structures comprised of multiple data objects, when an equal comparison of the value of one data object does not necessarily mean that the whole data structure is unchanged. In that case, finding an equal value for one data object may not mean that you are operating on the intended data structure element; other data objects in the data structure may have been independently changed.

A similar example is that CMPSW might not be sufficient for dequeueing an element from a queue residing in shared storage. More specifically, consider a queue that has an identified "head" offset value and some number of elements that each include a "next" offset field (a binary data object). Just because the element currently at the head of the queue has the same offset value as was previously copied to

local storage for use in a CMPSW does not mean that the element's current value of ″next″ is the same as the ″next″ value copied to local storage when the ″head″ offset value was copied. That is, multiple elements may have been dequeued and the element with the saved ″head″ offset, but a different ″next″ value, may have been enqueued since the local copies were made. Setting the new queue head value with the copied ″next″ value would thus be invalid. Update of the queue data structure clearly has not been properly synchronized by such a use of CMPSW. Instead, a CMPSW of a data object that consists of two smaller data objects: an offset and a use count that is incremented every time the element is used, would be a safe approach if the use count values will always be different.

Resultant Conditions:

•

- Equal-The first compare operand is equal to the second compare operand.
- Unequal -The first compare operand is unequal to the second compare operand.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compare and Swap (CMPSW)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4 [5]
CMPSWB 1C37	Branch options	Compare operand 1	Compare operand 2	Swap operand	Branch target
CMPSWI 1837	Indicator options	Compare operand 1	Compare operand 2	Swap operand	Indicator target

Operand 1: Character(1,2,4,8) variable scalar.

Operand 2: Character(1,2,4,8) variable scalar.

Operand 3: Character(1,2,4,8) scalar.

Operand 4 [4-5]:

-
- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CMPSWP is 156.
CMPSWP (
        op1   : address of a scalar(1,2,4,8) value
        op2   : address of a scalar(1,2,4,8) value (has alignment
                restrictions based on the length of the scalar - see
                description below)
        op3   : scalar(1,2,4,8) value
        cntl  : signed binary(4) literal value   (this operand
                is optional -- see description below)
) :   signed binary(4)
```

If the values of op1 and op2 are equal, the value 1 is returned. Otherwise, the value 0 is returned.

Description: The value of the first compare operand is compared with the value of the second compare operand. If they are equal, the *swap* operand is stored in the second compare operand's location. If they are unequal, the second compare operand is stored into the first compare operand's location. Based on the comparison, the resulting condition is used with the extender field to:

•

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

When an equal comparison occurs, it is assured that no access by another Compare and Swap instruction will occur at the second compare operand location between the moment that the second compare operand is fetched for comparison and the moment that the *swap* operand is stored at the second compare operand location.

When an unequal comparison occurs, no atomicity guarantees are made regarding the store to the first compare operand location and other Compare and Swap instruction access. Thus only the second compare operand should be a variable shared for concurrent processing control.

Both compare operands must be 1, 2, 4 or 8 byte character variable scalars and the *swap* operand must be a 1, 2, 4 or 8 byte character scalar. All three operands must have the same length. Failure to have the operands the same length will not be detected and the results of the Compare and Swap instruction are undefined when this occurs. The second operand must be aligned based on its length:

•

- one byte length - no alignment restrictions
- two byte length - halfword aligned
- four byte length - fullword aligned
- eight byte length - doubleword aligned

Failure to have the second operand aligned properly will not be detected, but the results of the Compare and Swap instruction are undefined when this occurs.

For bound program access, the *cntl* operand is optional, and does not need to be specified. If the *cntl* operand is specified, it has the following effect depending on the value:

•

- Hex 0x0 = (Default) Storage synchronization is performed both before and after a successful store of the *swap* operand.
- Hex 0x1 = No storage synchronization is performed

If the *cntl* operand is not specified, the instruction proceeds as if the default value had been specified.

The machine does not enforce that only the values enumerated above are used for the *cntl* operand. When specified, use of any value other than those enumerated above may result in unpredictable behavior.

Warning: Temporary Level 3 Header

Storage Synchronization

Unless explicitly disabled with the *no storage synchronization is performed* control option (disabling is possible only using bound program access), this instruction synchronizes storage both before and after the *swap* operand is stored to *op2*. Synchronized operation provides the following guarantees:

•

- When reading shared storage after a successful store of the *swap* operand, the thread performing the CMPSW will have a view of shared storage no less current then the most recent synchronizing actions taken by threads writing the shared storage.
- Shared data written by the thread performing the CMPSW prior to a successful store of the *swap* operand will be current from the perspective of other threads at their next synchronizing action.

When synchronization is disabled for CMPSW, storage synchronization can be accomplished using other storage synchronizing MI instructions, such as SYNCSTG.

See Storage Synchronization Concepts for additional information on storage synchronization.

Usage Note

Note that the compare and swap construct (including both this CMPSW instruction and support found on other architectures) simply compares values; it cannot determine whether the presence of the same value indicates that the data object for which an update is being synchronized is in fact the intended data object.

For example, CMPSW is insufficient for modifications to data structures comprised of multiple data objects, when an equal comparison of the value of one data object does not necessarily mean that the whole data structure is unchanged. In that case, finding an equal value for one data object may not mean that you are operating on the intended data structure element; other data objects in the data structure may have been independently changed.

A similar example is that CMPSW might not be sufficient for dequeueing an element from a queue residing in shared storage. More specifically, consider a queue that has an identified ″head″ offset value and some number of elements that each include a ″next″ offset field (a binary data object). Just because the element currently at the head of the queue has the same offset value as was previously copied to local storage for use in a CMPSW does not mean that the element’s current value of ″next″ is the same as the ″next″ value copied to local storage when the ″head″ offset value was copied. That is, multiple elements may have been dequeued and the element with the saved ″head″ offset, but a different ″next″ value, may have been enqueued since the local copies were made. Setting the new queue head value with the copied ″next″ value would thus be invalid. Update of the queue data structure clearly has not been properly synchronized by such a use of CMPSW. Instead, a CMPSW of a data object that consists of two smaller data objects: an offset and a use count that is incremented every time the element is used, would be a safe approach if the use count values will always be different.

Resultant Conditions:

•

- Equal-The first compare operand is equal to the second compare operand.
- Unequal -The first compare operand is unequal to the second compare operand.

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compare Bytes Left-Adjusted (CMPBLA)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4, 5]
CMPBLAB 1CC2	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPBLAI 18C2	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Numeric scalar or character scalar.

Operand 2: Numeric scalar or character scalar.

Operand 3 [4, 5]:

•

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction compares the logical string values of two left-adjusted compare operands. The logical string value of the first compare operand is compared with the logical string value of the second compare operand (no padding done). Based on the comparison, the resulting condition is used with the extender field to:

•

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The compare operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The compare operands are compared byte by byte, from left to right with no numeric conversions performed. The length of the operation is equal to the length of the shorter of the two compare operands. The comparison begins with the leftmost byte of each of the compare operands and proceeds until all bytes of the shorter compare operand have been compared or until the first unequal pair of bytes is encountered.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either or both compare operands is that the instruction's resultant condition is *equal*.

Resultant Conditions: The scalar first compare operand's string value is one of the following as compared to the second compare operand.

•
- Higher
- Lower
- Equal

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare Bytes Left-Adjusted with Pad (CMPBLAP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4 [5, 6]
CMPBLAPB 1CC3	Branch options	Compare operand 1	Compare operand 2	Pad	Branch targets
CMPBLAPI 18C3	Indicator options	Compare operand 1	Compare operand 2	Pad	Indicator targets

Operand 1: Numeric scalar or character scalar.

Operand 2: Numeric scalar or character scalar.

Operand 3: Numeric scalar or character scalar.

Operand 4 [5, 6]:

•

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction compares the logical string values of two left-adjusted compare operands (padded if needed). The logical string value of the first compare operand is compared with the logical string value of the second compare operand. Based on the comparison, the resulting condition is used with the extender field to:

•

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The compare operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The compare operands are compared byte by byte, from left to right with no numeric conversions being performed.

The length of the operation is equal to the length of the longer of the two compare operands. The shorter of the two compare operands is logically padded on the right with the 1-byte value indicated in the *pad* operand. If the *pad* operand is more than 1 byte in length, only its leftmost byte is used. The comparison begins with the leftmost byte of each of the compare operands and proceeds until all the bytes of the longer of the two compare operands have been compared or until the first unequal pair of bytes is encountered. All excess bytes in the longer of the two compare operands are compared to the pad value.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for one of the compare operands is that the other compare operand is compared with an equal length string of pad character values. When a null substring reference is specified for both compare operands, the resultant condition is *equal*.

Substring operand references that allow for a null substring reference (a length value of zero) may not be specified for operand 3.

Resultant Conditions: The scalar first compare operand's string value is one of the following as compared to the second compare operand.

•
- Higher
- Lower
- Equal

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare Bytes Right-Adjusted (CMPBRA)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4, 5]
CMPBRAB 1CC6	Branch options	Compare operand 1	Compare operand 2	Branch targets

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4, 5]
CMPBRAI 18C6	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Numeric scalar or character scalar.

Operand 2: Numeric scalar or character scalar.

Operand 3 [4, 5]:

•

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction compares the logical string values of two right-adjusted compare operands. The logical string value of the first compare operand is compared with the logical string value of the second compare operand (no padding done). Based on the comparison, the resulting condition is used with the extender field to:

•

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The compare operands can be either string or numeric. Any numeric operands are interpreted as logical character strings.

The compare operands are compared byte by byte, from left to right with no numeric conversions performed. The length of the operation is equal to the length of the shorter of the two compare operands. The comparison begins with the leftmost byte of each of the compare operands and proceeds until all bytes of the shorter compare operand have been compared or until the first unequal pair of bytes is encountered.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either or both compare operands is that the instruction's resultant condition is *equal*.

Resultant Conditions: The scalar first compare operand's string value is one of the following as compared to the second compare operand.

•

- Higher
- Lower
- Equal

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2C Program Execution

- 2C04 Branch Target Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare Bytes Right-Adjusted with Pad (CMPBRAP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4 [5, 6]
CMPBRAPB 1CC7	Branch options	Compare operand 1	Compare operand 2	Pad	Branch targets
CMPBRAPI 18C7	Indicator options	Compare operand 1	Compare operand 2	Pad	Indicator targets

Operand 1: Numeric scalar or character scalar.

Operand 2: Numeric scalar or character scalar.

Operand 3: Numeric scalar or character scalar.

Operand 4 [5, 6]:

•

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction compares the logical string values of the right-adjusted compare operands (padded if needed). The logical string value of the first compare operand is compared with the logical string value of the second compare operand. Based on the comparison, the resulting condition is used with the extender field to:

•

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The compare operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The compare operands are compared byte by byte, from left to right with no numeric conversions performed.

The length of the operation is equal to the length of the longer of the two compare operands. The shorter of the two compare operands is logically padded on the left with the 1-byte value indicated in the *pad* operand. If the *pad* operand is more than 1 byte in length, only its leftmost byte is used. The comparison begins with the leftmost byte of the longer of the compare operands. Any excess bytes (on the left) in the longer compare operand are compared with the pad value. All other bytes are compared with the corresponding bytes in the other compare operand. The operation proceeds until all bytes in the longer operand are compared or until the first unequal pair of bytes is encountered.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for one of the compare

operands is that the other compare operand is compared with an equal length string of pad character values. When a null substring reference is specified for both compare operands, the resultant condition is *equal*.

Substring operand references that allow for a null substring reference (a length value of zero) may not be specified for operand 3.

Resultant Conditions: The scalar first compare operand's string value is one of the following as compared to the second compare operand.

•
- Higher
- Lower
- Equal

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare Null-Terminated Strings Constrained (STRNCMPNULL)

Bound program access
Built-in number for STRNCMPNULL is 19. STRNCMPNULL (null_terminated_string1 : address of aggregate(*) null_terminated_string2 : address of aggregate(*) maximum_compare_length : unsigned binary(4) value which specifies the maximum number of bytes to compare) : signed binary(4) value which indicates if null_terminated_string1 is lexically less than (-1), equal to (0) or greater than (1) null_terminated_string2

Description: A compare is done of the storage specified by *null terminated string1* and *null terminated string2*. If the first byte of *null terminated string1* is less than the first byte of *null terminated string2*, the function returns -1; if the byte is greater the function returns 1. If the bytes are equal the function continues with the next byte. This process is repeated until either:

•

- The end of *null terminated string1* (ie. a null character) and *null terminated string2* is reached. The function returns 0.
- The end of *null terminated string1* is reached. The function returns -1.
- The end of *null terminated string2* is reached. The function returns 1.
- The number of characters specified by *maximum compare length* have been compared. The function returns 0.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare Numeric Value (CMPNV)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4-6]
CMPNVB 1C46	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPNVI 1846	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Numeric scalar.

Operand 2: Numeric scalar.

Operand 3 [4-6]:

•

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The numeric value of the first compare operand is compared with the signed or unsigned numeric value of the second compare operand. Based on the comparison, the resulting condition is used with the extender field to:

•
- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

Both the compare operands must be numeric with any implicit conversions occurring according to the rules of arithmetic operations as outlined in Arithmetic Operations. For a decimal operation, alignment of the assumed decimal point takes place by padding with 0's on the right end of the compare operand with lesser precision.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

When both operands are signed numeric or both are unsigned numeric, the length of the operation is equal to the length of the longer of the two compare operands. The shorter of the two operands is adjusted to the length of the longer operand according to the rules of arithmetic operations outlined in Arithmetic Operations.

When one operand is signed numeric and the other operand unsigned numeric, the unsigned operand is converted to a signed value with more precision than its current size. The length of the operation is equal to the length of the longer of the two compare operands. A negative signed numeric value will always be less than a positive unsigned value.

Floating-point comparisons use exponent comparison and significand comparison. For a denormalized floating-point number, the comparison is performed as if the denormalized number had first been normalized.

For floating-point, two values compare unordered when at least one comparand is NaN. Every NaN compares unordered with everything including another NaN value.

Floating-point comparisons ignore the sign of zero. Positive zero always compares equal with negative zero.

A *floating-point invalid operand* (hex 0C09) exception is signaled when two floating-point values compare unordered and no branch or indicator option exists for any of the unordered, negation of unordered, equal, or negation of equal resultant conditions.

When a comparison is made between a floating-point compare operand and a fixed-point decimal compare operand that contains fractional digit positions, a *floating-point inexact result* (hex 0C0D) exception may be signaled because of the implicit conversion from decimal to floating-point.

Resultant Conditions:

•
- High-The first compare operand has a higher numeric value than the second compare operand.
- Low-The first compare operand has a lower numeric value than the second compare operand.
- Equal-The first compare operand has a numeric value equal to the second compare operand.
- Unordered-The first compare operand is unordered compared to the second compare operand.

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C03 Decimal Point Alignment

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare Pointer for Object Addressability (CMPPTRA)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4]
CMPPTRAB 1CD2	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPPTRAI 18D2	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Data pointer, space pointer, system pointer, instruction pointer, or label pointer.

Operand 2: Data pointer, space pointer, system pointer, instruction pointer, or label pointer.

Operand 3 [4]:

-
- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CMPPTRA is 139.
CMPPTRA (
         compare_operand1   : space pointer(16) OR
                              data pointer OR
                              system pointer OR
                              label pointer
         compare_operand2   : space pointer(16) OR
                              data pointer OR
                              system pointer OR
                              label pointer
) :   signed binary(4) /* return_code */
```

The return code will be set as follows:

Return code	Meaning
1	Pointers address same object.
0	Pointers address different objects.

This built-in function is used to provide support for the branch and indicator forms of the CMPPTRA operation. The user must specify code to process the *return code* and perform the desired branching or indicator setting.

Description: The object addressed by operand 1 is compared with the object addressed by operand 2 to determine if both operands are addressing the same object. Based on the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form).

If operand 1 is a data pointer, a space pointer, or a system pointer, operand 2 may be a data pointer, a space pointer, or a system pointer in any combination. An *equal* condition occurs if the pointers are addressing the same object. For space pointers and data pointers, only the space they are addressing is considered in the comparison. That is, the space offset portion of the pointer is ignored. All implicit process spaces and teraspace are considered part of an active PCS (Process Control Space) object. Thus a pointer to teraspace or to an implicit process space addresses the same object as any other pointer which addresses the PCS of the process which contains the currently executing thread. Further, since any teraspace reference is local to a process, any two pointers to teraspace used within the same process are defined to address the same object.

For system pointer compare operands, an *equal* condition occurs if the system pointer is compared with a space pointer or data pointer that addresses the space that is associated with the object that is addressed by the system pointer. For example, a space pointer that addresses a byte in a space associated with a system object compares equal with a system pointer that addresses the system object.

For instruction pointer comparisons, both operands must be instruction pointers; otherwise, a *pointer type invalid* (hex 2402) exception is signaled. An *equal* condition occurs when both instruction pointers are addressing the same instruction in the same program. A *not equal* condition occurs if the instruction pointers are not addressing the same instruction in the same program.

For label pointer comparisons, both operands must be label pointers; otherwise, a *pointer type invalid* (hex 2402) exception is signaled. An *equal* condition occurs when both label pointers are addressing the same label in the same procedure. A *not equal* condition occurs if the label pointers are not addressing the same label in the same procedure.

A *pointer does not exist* (hex 2401) exception is signaled if a pointer does not exist in either of the operands.

Resultant Conditions:

•
- Equal
- Not equal

Warning: Temporary Level 3 Header

Authorization Required

•
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•
- Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare Pointer for Space Addressability (CMPPSPAD)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4-6]
CMPPSPADB 1CE6	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPPSPADI 18E6	Indicator options	Compare Operand 1	Compare Operand 2	Indicator targets

Operand 1: Space pointer or data pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, space pointer, or data pointer.

Operand 3 [4-6]:

•

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The space addressability contained in the pointer specified by operand 1 is compared with the space addressability defined by operand 2.

The value of the operand 1 pointer is compared based on the following:

•

- If operand 2 is a scalar data object (element or array), the space addressability of that data object is compared with the space addressability contained in the operand 1 pointer.
- If operand 2 is a pointer, it must be a space pointer or data pointer, and the space addressability contained in the pointer is compared with the space addressability contained in the operand 1 pointer.

Based on the results of the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form). If the operands are not in the same space, the resultant condition is *unequal*. If the operands are in the same space and the offset into the space of operand 1 is larger or smaller than the offset of operand 2, the resultant condition is *high* or *low*, respectively. An *equal* condition occurs only if the operands are in the same space at the same offset. Therefore, the resultant conditions (*high, low, equal*, and *unequal*) are mutually exclusive. Consequently, if you specify that an action be taken upon the nonexistence of a condition, this results in the action being taken upon the occurrence of any of the other three possible conditions. For example, a branch not high would result in the branch being taken on a *low, equal*, or *unequal* condition.

The *object destroyed* (hex 2202) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 or operand 2 is a space pointer machine object or when operand 2 is a scalar based on a space pointer machine object. This occurs when the space pointer machine object contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, the resulting condition of the comparison operation is not defined other than that it will be one of the four valid resultant conditions for this instruction.

When the Override Program Attributes (OVRPGATR) instruction is used to override this instruction, the *pointer does not exist* (hex 2401) exception is not signaled when operand 1 or operand 2 is a space pointer (i.e. either a space pointer data object or a space pointer machine object). Furthermore, some comparisons involving space pointers are defined even when one or both of the compare operands is a pointer subject to the pointer does not exist condition. Specifically, if both compare operands are subject to the pointer does not exist condition, the resultant condition is *equal*. When one space pointer is set and one is subject to the pointer does not exist condition, the resultant condition is *unequal*, but undefined with respect to comparisons which include specification of the *high* or *low* conditions.

Resultant Conditions:

•

- High
- Low
- Equal
- Unequal

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare Pointer Type (CMPPTRT)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4]
CMPPTRTB 1CE2	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPPTRTI 18E2	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Data pointer, space pointer, system pointer, instruction pointer, invocation pointer, procedure pointer, label pointer, suspend pointer, synchronization pointer, object pointer, or field pointer.

Operand 2: Character(1) scalar or null.

Operand 3 [4]:

•

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CMPPTRT is 140.
CMPPTRT (
         pointer_type   : aggregate(1,2,4) OR
                          signed binary(1,2,4) OR
                          unsigned binary(1,2,4)
         pointer        : pointer(16)
) :   signed binary(4) /* return_code */
```

The return code will be set as follows:

Return code	Meaning
1	Pointer is of specified type.
0	Pointer is not of specified type.

The *pointer type* operand corresponds to operand 2 on the CMPPTRT operation. The *pointer* operand corresponds to operand 1 on the CMPPTRT operation.

This built-in function is used to provide support for the branch and indicator forms of the CMPPTRT operation. The user must specify code to process the *return code* and perform the desired branching or indicator setting.

Description: The instruction compares the pointer type currently in operand 1 with the character scalar identified by operand 2. Based on the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form).

Operand 1 can specify a space pointer machine object only when operand 2 is null.

An unresolved operand 1 pointer is not resolved by this instruction.

If operand 2 is null or if operand 2 specifies a comparison value of hex 00, an equal condition occurs if a pointer does not exist in the storage area identified by operand 1.

Following are the allowable values for operand 2:

Hex 00 - A pointer does not exist at this location

Hex 01 - System pointer

Hex 02 - Space pointer

Hex 03 - Data pointer

Hex 04 - Instruction pointer

Hex 05 - Invocation pointer

Hex 06 - Procedure pointer

Hex 07 - Label pointer

Hex 08 - Suspend pointer

Hex 09 - Synchronization pointer

Hex 0A - Object pointer

Hex 0B - Field pointer

Resultant Conditions:

•
• Equal
• Not equal

Warning: Temporary Level 3 Header

Authorization Required

•
• None

Lock Enforcement

•
• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare Pointers for Equality (CMPPTRE)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4]
CMPPTREB 1C12	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPPTREI 1812	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Pointer data object

Operand 2: Pointer data object

Operand 3 [4]:

•

• *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.

- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The pointer specified by operand 1 is compared with the pointer specified by operand 2 to determine if both operands are of the same type and contain equal values. Based on the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form).

Pointers may be specified for operands 1 and 2 in any combination. An *equal* condition occurs if the pointers are of the same type and contain the same value, or if neither pointer has been set. If one pointer is set and the other is not, a *not equal* condition occurs.

System pointers and data pointers are not resolved by this instruction. The comparison result is undefined when an unresolved pointer is supplied for one or both operands.

Note that any authorities stored in a resolved system pointer are part of the pointer. Thus system pointers pointing to the same object, but with different levels of authority, will compare as *not equal*.

Since any pointer type may be specified for this instruction, the *pointer type invalid* (hex 2402) exception is not signaled except for pointers used as a base for the operands. Similarly, since the instruction accepts unset pointers, the *pointer does not exist* (hex 2401) exception is not signaled except for pointers used as a base for the operands.

While it is possible for two synchronization pointers to compare as equal, the item referenced by the pointer may not be usable from both pointer locations. A copied synchronization pointer is not useful, because a synchronization pointer is defined to reside at only a single location in memory.

Resultant Conditions:

-
- Equal
- Not equal

Warning: Temporary Level 3 Header

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

Compare Space Addressability (CMPSPAD)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4-6]
CMPSPADB 1CF2	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPSPADI 18F2	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, pointer data object, pointer data object array.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, pointer data object, pointer data object array.

Operand 3 [4-6]:

-
- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The space addressability of the object specified by operand 1 is compared with the space addressability of the object specified by operand 2.

Based on the results of the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form). If the operands are not in the same space, the resultant condition is *unequal*. If the operands are in the same space and the offset of operand 1 is larger or smaller than the offset of operand 2, the resultant condition is *high* or *low*, respectively. Equal occurs only if the operands are in the same space at the same offset. Therefore, the resultant conditions (*high, low, equal*, and *unequal*) are mutually exclusive. Consequently, if you specify that an action be taken upon the nonexistence of a condition, this results in the action being taken upon the occurrence of any of the other three possible conditions. For example, a branch not high would result in the branch being taken on a *low, equal*, or *unequal* condition.

If a pointer data object operand contains a data pointer value upon execution of the instruction, the addressability is compared to the pointer data object rather than to the scalar described by the data pointer value. The variable scalar references allowed on operands 1 and 2 cannot be described through a data pointer value.

The *object destroyed* (hex 2202) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 or operand 2 is based on a space pointer machine object. This occurs when the space pointer machine object contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, the resulting condition of the comparison operation is not defined other than that it will be one of the four valid resultant conditions for this instruction.

Resultant Conditions:

-
- High
- Low
- Equal
- Unequal

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compare To Pad (CMPTOPAD)

Bound program access

```
Built-in number for CMPTOPAD is 429.
CMPTOPAD (
          string          : address of aggregate(*)
          pad             : signed binary(1,4,8) - rightmost byte specifies
                                                   the pad value  OR
                            unsigned binary(1)  OR
                            aggregate(1)
          string_length   : unsigned binary(4,8) value which specifies the
                            length of the string
) :   signed binary(4,8) value which indicates if the string is
      lexically less than (-1), equal to (0) or greater than (1) a
      string of equal length which would be composed entirely of the pad
      byte value, replicated for the length of the string
```

Description: A logical (character) compare is done between the storage specified by *string* and the *pad* byte, which is logically replicated as necessary. If the first byte of the *string* is less than the *pad* byte value, the result is -1; if the string byte is greater the result is 1. If the bytes are equal the operation continues with the next byte of the *string.* This process is repeated until an inequality result is returned, or the number of bytes specified by *string length* have been compared. If all bytes compare equal, the result is 0. If the *string length* has a value of zero, the result is 0.

The *string* operand can point to storage containing values of any data type. The values will be interpreted as a logical character string. If the *string* operand points to storage which contains pointers, any pointer tags will not be taken into account, since this is strictly a byte comparison.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Complement String (COMSTR)

Bound program access

```
Built-in number for COMSTR is 452.
COMSTR (
        receiver_string   : address of aggregate(*) for the result of
                            the complement
        source_string     : address of aggregate(*)
        string_length     : unsigned binary(4,8) value which specifies the
                            length of the two strings
)
```

Description: Each byte value of the *source string*, for the number of bytes indicated by *string length,* is logically complemented on a bit-by-bit basis (i.e. one's complement). The results are placed in the *receiver string*. If the strings overlap in storage, predictable results occur only if the overlap is fully coincident.

If the space(s) indicated by the two addresses are not long enough to contain the number of bytes indicated by *string length,* a *space addressing violation* (hex 0601) is signalled. Partial results in this case are unpredictable.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compress Data (CPRDATA)

Op Code (Hex)	Operand 1
1041	Compress data template

Operand 1: Space pointer.

Bound program access
Built-in number for CPRDATA is 107. CPRDATA (compress_data_template : address)

Description: The instruction compresses user data of a specified length. Operand 1 identifies a template which identifies the data to be compressed. The template also identifies the result space to receive the compressed data.

The *compress data template* must be aligned on a 16-byte boundary. The format is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Source length	Bin(4)
4	4	Result area length	Bin(4)
8	8	Actual result length	Bin(4) +
12	C	Compression algorithm	Bin(2)
		1 = Simple TERSE algorithm	
		2 = IBM(R) LZ1 algorithm	
14	E	Reserved (binary 0)	Char(18)
32	20	Source space pointer	Space pointer
48	30	Result space pointer	Space pointer
64	40	—- End —-	

Note: The input value associated with template fields annotated with a plus sign (+) are ignored by the instruction; these fields are updated by the instruction to return information about instruction execution.

The data at the location specified by the **source space pointer** for the length specified by the **source length** is compressed and stored at the location specified by the **result space pointer**. The **actual result length** is set to the number of bytes in the compressed result. The source data is not modified.

The value of both the *source length* field and *result area length* field must be greater than or equal to zero. If either of these conditions is not met, a *template value invalid* (hex 3801) exception is signalled. If the length of the compressed result is greater than the value in the *result area length* field, a *materialization length invalid* (hex 3803) exception is signalled.

The **compression algorithm** field specifies the algorithm used to compress the data. The *IBM LZ1 algorithm* tends to produce better compression on shorter input strings than the *simple TERSE algorithm*. The algorithm choice is stored in the compressed output data so the Decompress Data (DCPDATA) instruction will automatically select the correct decompression algorithm.

Only scalar (non-pointer) data is compressed, so any pointers in the data to be compressed are destroyed in the output of the Compress Data instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compute Array Index (CAI)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
1044	Array index	Subscript A	Subscript B	Dimension

Operand 1: Binary(2) variable scalar.

Operand 2: Binary(2) scalar.

Operand 3: Binary(2) scalar.

Operand 4: Binary(2) constant scalar object or immediate operand.

Description: This instruction provides the ability to reduce multidimensional array subscript values into a single index value which can then be used in referencing the single-dimensional arrays of the system. This index value is computed by performing the following arithmetic operation on the indicated operands.

Array Index = Subscript A + ((Subscript B-1) X Dimension)

The numeric value of the *subscript B* operand is decreased by 1 and multiplied by the numeric value of the *dimension* operand. The result of this multiplication is added to the *subscript A* operand and the sum is placed in the *array index* operand.

All the operands must be binary with any implicit conversions occurring according to the rules of arithmetic operations documented in Arithmetic Operations. The usual rules of algebra are observed concerning the subtraction, addition, and multiplication of operands.

This instruction provides for mapping multidimensional arrays to single-dimensional arrays. The elements of an array with the dimensions (d1, d2, d3, ..., dn) can be defined as a single-dimensional array with d1*d2*d3*...*dn elements. To reference a specific element of the multidimensional array with subscripts (s1,s2,s3,...sn), it is necessary to convert the multiple subscripts to a single subscript for use in the single-dimensional array. This single subscript can be computed using the following:

```
s1+((s2-1)*d1)+(s3-1)*d1*d2)+...+((sn-1)*d*d2*d3*...*dm)
```

where m = n-1

The CAI instruction is used to form a single index value from two subscript values. To reduce N subscript values into a single index value, N-1 uses of this instruction are necessary.

Assume that S1, S2, and S3 are three subscript values and that D1 is the size of one dimension, D2 is the size of the second dimension, and the D1D2 is the product of D1 and D2. The following two uses of this instruction reduce the three subscripts to a single subscript.

```
CAI INDEX, S1, S2, D1      Calculates s1+(s2-1)*d1
CAI INDEX, INDEX, S3, D1D2 Calculates s1+(s2-1)*d1+(s3-1)*d2*d1
```

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C0A Size

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compute Date Duration (CDD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0424	Date duration	Date 1	Date 2	Instruction template

Operand 1: Packed decimal variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4: Space pointer.

Bound program access

```
Built-in number for CDD is 101.
CDD (
     date_duration          : address of packed decimal
     date1                  : address
     date2                  : address
     instruction_template   : address
)
```

Description: The date specified by operand 3 is subtracted from the date specified by operand 2 and the resulting *date duration* is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

A negative value will be returned when the *date 1* operand is less than the *date 2* operand.

The following describes the *instruction template*.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Instruction template	Char(*)	
0	0		Instruction template size	UBin(4)
4	4		Operand 1 data definitional attribute template number	UBin(2)
6	6		Operand 2 data definitional attribute template number	UBin(2)
8	8		Operand 3 data definitional attribute template number	UBin(2)
10	A		Reserved (binary 0)	Char(2)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
12	C		Operand 2 length	UBin(2)
14	E		Operand 3 length	UBin(2)
16	10		Reserved (binary 0)	Char(26)
42	2A		Data definitional attribute template list	Char(*)
42	2A		Size of the DDAT list	UBin(4)
46	2E		Number of DDATs	UBin(2)
48	30		Reserved (binary 0)	Char(10)
58	3A		DDAT offset	[*] UBin(4)
*	*		Data definitional attribute template	[*] Char(*)
*	*	— End —		

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDAT for operand 1 must be valid for a date duration. The DDATs for operands 2 and 3 must be valid for a date and must be identical. Otherwise, a *template value invalid* (hex 3801) exception will be signaled.

Operand 2 length and **operand 3 length** are specified in number of bytes.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the template. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C15 Date Boundary Overflow

0C16 Data Format Error

0C17 Data Value Error

0C18 Date Boundary Underflow

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compute Length of Null-Terminated String (STRLENNULL)

Bound program access
Built-in number for STRLENNULL is 23. STRLENNULL (null_terminated_string : address of aggregate(*)) : unsigned binary(4) value which specifies the number of bytes between the beginning of the string and the first null (ie. zero) byte

Description: The *null terminated string* is searched for the first null byte (hex 00). The number of non-null bytes found is returned.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Compute Math Function Using One Input Value (CMF1)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CMF1 100B		Receiver	Controls	Source	
CMF1B 1C0B	Branch options	Receiver	Controls	Source	Branch targets
CMF1I 180B	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Numeric variable scalar.

Operand 2: Character(2) scalar.

Operand 3: Numeric scalar.

Operand 4-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The mathematical function, indicated by the *controls* operand, is performed on the *source* operand value and the result is placed in the *receiver* operand.

The calculation is always done in floating-point.

The result of the operation is copied into the *receiver* operand.

The *controls* operand must be a character scalar that specifies which mathematical function is to be performed. It must be at least 2 bytes in length and has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Controls operand	Char(2)

Hex 0001 = Sine

Hex 0002 = Arc sine

Hex 0003 = Cosine

Hex 0004 = Arc cosine

Hex 0005 = Tangent

Hex 0006 = Arc tangent

Hex 0007 = Cotangent

Hex 0010 = Exponential function

Hex 0011 = Logarithm based e (natural logarithm)

Hex 0012 = Sine hyperbolic

Hex 0013 = Cosine hyperbolic

Hex 0014 = Tangent hyperbolic

Hex 0015 = Arc tangent hyperbolic

Hex 0020 = Square root

All other values are reserved

Offset			
Dec	Hex	Field Name	Data Type and Length
2	2	— End —	

The *controls* operand mathematical functions are as follows:

•

- Hex 0001-Sine

  The sine of the numeric value of the *source* operand, whose value is considered to be in radians, is computed and placed in the *receiver* operand.

  The result is in the range:

  ```
  -1 <= SIN(x) <= 1
  ```

- Hex 0002-Arc sine

  The arc sine of the numeric value of the *source* operand is computed and the result (in radians) is placed in the *receiver* operand.

  The result is in the range:

  ```
  -pi/2 <= ASIN(x) <= +pi/2
  ```

- Hex 0003-Cosine

  The cosine of the numeric value of the *source* operand, whose value is considered to be in radians, is computed and placed in the *receiver* operand.

  The result is in the range:

  ```
  -1 <= COS(x) <= 1
  ```

- Hex 0004-Arc cosine

  The arc cosine of the numeric value of the *source* operand is computed and the result (in radians) is placed in the *receiver* operand.

  The result is in the range:

  ```
  0 <= ACOS(x) <= pi
  ```

- Hex 0005-Tangent

  The tangent of the *source* operand, whose value is considered to be in radians, is computed and the result is placed in the *receiver* operand.

  The result is in the range:

  ```
  -infinity <= TAN(x) <= +infinity
  ```

- Hex 0006-Arc tangent

  The arc tangent of the *source* operand is computed and the result (in radians) is placed in the *receiver* operand.

  The result is in the range:

  ```
  -pi/2 <= ATAN(x) <= pi/2
  ```

- Hex 0007-Cotangent

  The cotangent of the *source* operand, whose value is considered to be in radians, is computed and the result is placed in the *receiver* operand.

  The result is in the range:

  ```
  -infinity <= COT(x) <= +infinity
  ```

- Hex 0010-Exponential function

  The computation e power (*source* operand) is performed and the result is placed in the *receiver* operand.

  The result is in the range:

  ```
  0 <= EXP(x) <= +infinity
  ```

- Hex 0011-Logarithm based e (natural logarithm)

The natural logarithm of the *source* operand is computed and the result is placed in the *receiver* operand.

The result is in the range:

```
-infinity <= LN(x) <= +infinity
```

- Hex 0012-Sine hyperbolic

  The sine hyperbolic of the numeric value of the *source* operand is computed and the result (in radians) is placed in the *receiver* operand.

  The result is in the range:

  ```
  -infinity <= SINH(x) <= +infinity
  ```

- Hex 0013-Cosine hyperbolic

  The cosine hyperbolic of the numeric value of the *source* operand is computed and the result (in radians) is placed in the *receiver* operand.

  The result is in the range:

  ```
  +1 <= COSH(x) <= +infinity
  ```

- Hex 0014-Tangent hyperbolic

  The tangent hyperbolic of the numeric value of the *source* operand is computed and the result (in radians) is placed in the *receiver* operand.

  The result is in the range:

  ```
  -1 <= TANH(x) <= +1
  ```

- Hex 0015-Arc tangent hyperbolic

  The inverse of the tangent hyperbolic of the numeric value of the *source* operand is computed and the result (in radians) is placed in the *receiver* operand.

  The result is in the range:

  ```
  -infinity <= ATANH(x) <= +infinity
  ```

- Hex 0020-Square root

  The square root of the numeric value of the *source* operand is computed and placed in the *receiver* operand.

  The result is in the range:

  ```
  0 <= SQRT(x) <= +infinity
  ```

The following chart shows some special cases for certain arguments (X) of the different mathematical functions which take one argument value.

X	Masked	UnMasked					Maximum	Minimum	
Function	**NaN**	**NaN**	**+Infinity**	**-Infinity**	**+0**	**-0**	**Value**	**Value**	**Other**
Sine	g	A(e)	A(f)	A(f)	+0	-0	A(1,f)	A(1,f)	B(3)
Arc sine	g	A(e)	A(f)	A(f)	+0	-0	A(6,f)	A(6,f)	-
Cosine	g	A(e)	A(f)	A(f)	+1	+1	A(1,f)	A(1,f)	B(3)
Arc cosine	g	A(e)	A(f)	A(f)	+pi/2	+pi/2	A(6,f)	A(6,f)	-
Tangent	g	A(e)	A(f)	A(f)	+0	-0	A(1,f)	A(1,f)	B(3)
Arc Tangent	g	A(e)	+pi/2	-pi/2	+0	-0	-	-	-
Cotangent	g	A(e)	A(f)	A(f)	+inf	-inf	A(1,f)	A(1,f)	B(3)
E Exponent	g	A(e)	+inf	+0	+1	+1	C(4,a)	D(5,b)	-
Logarithm	g	A(e)	+inf	A(f)	-inf	-inf	-	-	A(2,f)
Sine hyperbolic	g	A(e)	+inf	-inf	+0	-0	-	-	-
Cosine hyperbolic	g	A(e)	+inf	+inf	+1	+1	-	-	-
Tangent hyperbolic	g	A(e)	+1	-1	+0	-0	-	-	-

X	Masked	UnMasked					Maximum	Minimum	
Function	**NaN**	**NaN**	**+Infinity**	**-Infinity**	**+0**	**-0**	**Value**	**Value**	**Other**
Arc tangent hyperbolic	g	A(e)	A(f)	A(f)	+0	-0	A(6,f)	A(6,f)	-
Square root	g	A(e)	+inf	A(f)	+0	-0	-	-	A(2,f)
Special cases for single argument math functions									

Capital letters in the chart indicate the exceptions, small letters indicate the returned results, and Arabic numerals indicate the limits of the arguments (X) as defined in the following lists:

A = *Floating-point invalid operand* (hex 0C09) exception (no result stored if unmasked; if masked, occurrence bit is set)

B = *Floating-point inexact result* (hex 0C0D) exception (result is stored whether or not exception is masked)

C = *Floating-point overflow* (hex 0C06) exception (no result is stored if unmasked; if masked, occurrence bit is set)

D = *Floating-point underflow* (hex 0C07) exception (no result is stored if unmasked; occurrence bit is always set)

a = Result follows the rules that depend on round mode

b = Result is +0 or a denormalized value

c = Result is +infinity

d = Result is -infinity

e = Result is the masked form of the input NaN

f = Result is the system default masked NaN

g = Result is the input NaN

inf = Result is infinity

1 = | pi * 2**50 |=Hex 432921FB54442D18

2 = Argument is in the range: -inf < x < -0

3 = | pi * 2**26 |=Hex 41A921FB54442D18

4 = 1n(2**1023) Hex 40862E42FEFA39EF

5 = 1n(2**-1021.4555)=Hex C086200000000000

6 = Argument is in the range: -1 <= x <= +1

The following chart provides accuracy data for the mathematical functions that can be invoked by this instruction.

Function Name	Sample Selection			Accuracy data			
				Relative Error(e)		Absolute Error(E)	
	A	**Range of x**	**D**	**MAX(e)**	**SD(e)**	**MAX(E)**	**SD(E)**
Arc cosine	9	0<=x<=3.14	U			8.26* 10**-14	2.11*10**-15
Arc sine	10	-1.57<=x<=1.57	U	1.02*10**-13	2.66*10**-15		
Arc tangent	1	-pi/2<x<pi/2	1			3.33*10**-16	9.57*10**-17
Arc tangent hyperbolic	14	-3<=x<=3	U			1.06*10**-14	1.79*10**-15
Cosine		(See Sine below)					
Cosine hyperbolic		(See Sine Hyperbolic below)					
Cotangent	11	-10<=x<=100	U	4.83*10**-16	1.48*10**-16		
		.000001<=x<=.001	U	4.36*10**-16	1.49*10**-16		
		4000<=x<=4000000	U	5.72*10**-16	1.46*10**-16		
Exponential	2	-100<=x<=300	U	5.70*10**-14	1.13*10**-14		
Natural Logarithm	3	0.5<=x<=1.5	U			2.77*10**-16	8.01*10**-17

Function Name	Sample Selection			Accuracy data			
				Relative Error(e)		Absolute Error(E)	
	A	Range of x	D	MAX(e)	SD(e)	MAX(E)	SD(E)
Natural Logarithm	4	-100<=x<=700	E	2.17*10**-16	7.37*10**-17		
Sine cosine	5	-10<=x<=100	U			2.22*10**-16	1.31*10**-16
		.000001<=x<=.001	U			2.22*10**-16	1.56*10**-16
		4000<=x<=4000000	U			2.22*10**-16	1.28*10**-16
Sine cosine	6	-10<=x<=100	U			3.33*10**-16	8.39*10**-17
		.000001<=x<=.001	U			4.33*10**-19	1.28*10**-19
		4000<=x<=4000000	U			3.33*10**-16	8.17*10**-17
Sine/cosine hyperbolic	12	-100<=x<=300	U	6.31*10**-16	1.97*10**-16		
Square root	7	-100<=x<=700	E	4.13*10**-16	1.27*10**-16		
Tangent	8	-10<=x<=100	U	4.59*10**-16	1.54*10**-16		
		.000001<=x<=.001	U	4.42*10**-16	1.44*10**-16	3.25*10**-19	8.06*10**-20
		4000<=x<=4000000	U	4.77*10**-16	1.43*10**-16		
Tangent hyperbolic	13	-100<=x<=300	U	8.35*10**-16	3.87*10**-17	2.22*10**-16	3.17*10**-17

Note:

Algorithm Notes:

1. f(x) = x, and g(x) = ATAN(TAN(x)).
2. f(x) = e**x, and g(x) = e**(1n(e**x)).
3. f(x) = 1n(x), and g(x) = 1n(e**(1n(x))).
4. f(x) = x, and g(x) = 1n(e**x).
5. Sum of squares algorithm. f(x) = 1, and g(x) = SIN(x))**2 + (COS(x))**2.
6. Double angle algorithm. f(x) - SIN(2x), and g(x) = 2*(SIN(x)*COS(x)).
7. f(x) = e(**x, and g(x) = (SQR(e**x))**2.
8. f(x) = TAN(x), and g(x) = SIN(x) / COS(x).
9. f(x) = x, and g(x) = ACOS(COS(x)).
10. f(x) = x, and g(x) = ASIN(SIN(x)).
11. f(x) = COT(x), and g(x) = COS(x) / SIN(x).
12. f(x) = SINH(2x), and g(x) = 2*(SINH(x)*COSH(x)).
13. f(x) = TANH(x), and g(x) = SINH(x) / COSH(x).
14. f(x) = x, and g(x) = ATANH(TANH(x)).

Distribution Note:

1. The sample input arguments were tangents of numbers, x, uniformly distributed between -pi/2 and +pi/2.

Accuracy data for CMF1 mathematical functions

The vertical columns in the accuracy data chart have the following meanings:

•

- *Function Name*: This column identifies the principal mathematical functions evaluated with entries arranged in alphabetical order by function name.
- *Sample Selection*: This column identifies the selection of samples taken for a particular math function through the following subcolumns:

  –

– *A*: identifies the algorithm used against the argument, x, to gather the accuracy samples. The numbers in this column refer to notes describing the functions, f(x) and g(x), which were calculated to test for the anticipated relation where f(x) should equal g(x). An accuracy sample then, is an evaluation of the degree to which this relation held true. The algorithm used to sample the arctangent function, for example, defines g(x) to first calculate the tangent of x to provide an appropriate distribution of input arguments for the arctangent function. Since f(x) is defined simply as the value of x, the relation to be evaluated is then x=ARCTAN(TAN(x)). This type of algorithm, where a function and its inverse are used in tandem, is the usual type employed to provide the appropriate comparison values for the evaluation.
– "Range of x": gives the range of x used to obtain the accuracy samples. The test values for x are uniformly distributed over this range. It should be noted that x is not always the direct input argument to the function being tested; it is sometimes desirable to distribute the input arguments in a nonuniform fashion to provide a more complete test of the function (see column D below). For each function, accuracy data is given for one or more segments within the valid range of x. In each case, the numbers given are the most meaningful to the function and range under consideration.
– *D*: identifies the distribution of arguments input to the particular function being sampled. The letter E indicates an exponential distribution. The letter U indicates a uniform distribution. A number refers to a note providing detailed information regarding the distribution.

• *Accuracy Data*: The maximum relative error and standard deviation of the relative error are generally useful and revealing statistics; however, they are useless for the range of a function where its value becomes zero. This is because the slightest error in the argument can cause an unpredictable fluctuation in the magnitude of the answer. When a small argument error would have this effect, the maximum absolute error and standard deviation of the absolute error are given for the range.
  –
  – *Relative Error (e)*: The maximum relative error and standard deviation (root mean square) of the relative error are defined:

MAX(e) = MAX(ABS((f(x) - g(x)) / f(x)))

where: MAX selects the largest of its arguments and ABS takes the absolute value of its argument.

SD(e) = SQR((1/N) SUMSQ((f(x) - g(x)) / f(x)))

where: SQR takes the square root of its argument and SUMSQ takes the summation of the squares of its arguments over all of the test cases.

  – *Absolute Error (E)*: The maximum absolute error produced during the testing and the standard deviation (root mean square) of the absolute error are:

MAX(E) = MAX(ABS(f(x) - g(x)))

where: the operators are those defined above.

SD(E) = SQR((1/N) SUMSQ(f(x) - g(x)))

where: the operators are those defined above.

Limitations (Subject to Change): The following are limits that apply to the functions performed by this instruction.

The *source* and *receiver* operands must both be specified as floating-point with the same length (4 bytes for short format or 8 bytes for long format).

Resultant Conditions:

•

• Positive-The algebraic value of the receiver operand is positive.

- Negative-The algebraic value of the receiver operand is negative.
- Zero-The algebraic value of the receiver operand is zero.
- Unordered-The value assigned to the floating-point result is NaN.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

2E02 Security Audit Journal Failure

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compute Math Function Using Two Input Values (CMF2)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-8]
CMF2 100C		Receiver	Controls	Source 1	Source 2	
CMF2B 1C0C	Branch options	Receiver	Controls	Source 1	Source 2	Branch targets
CMF2I 180C	Indicator options	Receiver	Controls	Source 1	Source 2	Indicator targets

Operand 1: Numeric variable scalar.

Operand 2: Character(2) scalar.

Operand 3: Numeric scalar.

Operand 4: Numeric scalar.

Operand 5-8:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The mathematical function, indicated by the *controls* operand, is performed on the source operand values and the result is placed in the *receiver* operand.

The calculation is always done in floating-point.

The *controls* operand must be a character scalar that specifies which mathematical function is to be performed. It must be at least 2 bytes in length and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Controls operand	Char(2)
		Hex 0001 = Power (x to the y)	
		All other values are reserved	
2	2	—- End —-	

The computation x power y, where x is the first source operand and y is the second source operand, is performed and the result is placed in the *receiver* operand.

The following chart shows some special cases for certain arguments of the power function (x**y). Within the chart, the capitalized letters X and Y refer to the absolute value of the arguments x and y; that is, X = |x| and Y = |y|.

y	-inf	y<0	y<0	y<0	-1	-1/2	+0	+1/2	+1	y>0	y>0	y>0	+inf	M-	UnM-
		y=	y=2n	real			or			y=	y=2n	real		NaN	NaN
x		2n+1					-0			2n+1					
+inf	+0	+0	+0	+0	+0	+0	+1	+inf	+inf	+inf	+inf	+inf	+inf	b	A(c)
x>1	+0	$\frac{+1}{X^{**}Y}$	$\frac{+1}{X^{**}Y}$	$\frac{+1}{X^{**}Y}$	$\frac{+1}{X}$	$\frac{+1}{SQRT(x)}$	+1	SQRT(x)	x	x**y	x**y	x**y	+inf	b	A(c)
X=+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	b	A(c)
0<x<1	+inf	$\frac{+1}{X^{**}Y}$	$\frac{+1}{X^{**}Y}$	$\frac{+1}{X^{**}Y}$	$\frac{+1}{X}$	$\frac{+1}{SQRT(X)}$	+1	SQRT(x)	x	x**y	x**y	x**y	+0	b	A(c)
x=+0	E(f)	E(f)	E(f)	E(f)	E(f)	E(f)	+1	+0	+0	+0	+0	+0	+0	b	A(c)
x=-0	E(f)	E(g)	E(f)	E(f)	E(g)	E(g)	+1	-0	-0	-0	+0	+0	+0	b	A(c)
0>x>-1	A(a)	$\frac{-1}{X^{**}Y}$	$\frac{+1}{X^{**}Y}$	A(a)	$\frac{-1}{X}$	A(a)	+1	A(a)	x	-X**y	X**y	A(a)	A(a)	b	A(c)
x=-1	A(a)	-1	+1	A(a)	-1	A(a)	+1	A(a)	-1	-1	+1	A(a)	A(a)	b	A(c)
x<-1	A(a)	$\frac{-1}{X^{**}Y}$	$\frac{+1}{X^{**}Y}$	A(a)	$\frac{-1}{X}$	A(a)	+1	A(a)	x	-X**y	X**y	A(a)	A(a)	b	A(c)

x=-inf	A(a)	-0	+0	A(a)	-0	A(a)	+1	A(a)	-inf	-inf	+inf	A(a)	A(a)	b	A(c)
Masked Nan	b	b	b	b	b	b	b	b	b	b	b	b	b	d	A(e)
Unmasked Nan	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(e)	A(e)
Special cases of the power function (xy)**															

Capital letters in the chart indicate the exceptions and small letters indicate the returned results as defined in the following list:

A	*Floating-point invalid operand* (hex 0C09) exception
E	*Floating-point zero divide* (hex 0C0E) exception
a	Result is the system default masked NaN
b	Result is the same NaN
c	Result is the same NaN masked
d	Result is one of the input NaNs
e	Result is a masked NAN
f	Result is +infinity
g	Result is -infinity

The following chart provides accuracy data for the mathematical function that can be invoked by this instruction.

+inf	+0	+0	+0	+0	+0	+0	+1	+inf	+inf	+inf	+inf	+inf	+inf	b	A(c)
x>1	+0	+1 —— x**Y	+1 —— x**Y	+1 —— x**Y	+1 —— x	+1 —— SQRT(x)	+1	SQRT(x)	x	x**y	x**y	x**y	inf	b	A(c)
x=+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	b	A(c)
0<x<1	+inf	+1/x**Y	+1/x**Y	+1/x**Y	+1/x	+1/SQRT(x)		x**y	x**y	x**y	+0	b	A(c)		
x=+0	E(f)	E(f)	E(f)	E(f)	E(f)	E(f)	+1	+0	+0	+0	+0	+0	+0	b	A(c)
Special cases of the power function															

Figure 1. Accuracy data for CMF2 mathematical functions.

Function Name	Sample Selection x	Sample Selection y	Accuracy Data MAX(e)	Accuracy Data SD(e)
Power	1/3	-345 <= y <= 330	4.99 * 10**-16	1.90 * 10**-16
	.75	-320 <= y <= 1320	2.96 * 10**-16	2.39 * 10**-16
	.9	-3605 <= y <= 3605	1.23 * 10**-16	1.02 * 10**-16
	10	-165 <= y <= 165	7.10 * 10**-16	3.18 * 10**-16
	712	-57 <= y <= 57	1.75 * 10**-15	7.24 * 10**-16

The vertical columns in the accuracy data chart have the following meanings:

•

- *Function Name*: This column identifies the mathematical function.
- *Sample Selection*: This column identifies the selection of samples taken for the power function. The algorithm used against the arguments, x and y, to gather the accuracy samples was a test for the anticipated relation where f(x) should equal g(x,y):

  where:

  ```
  f (x)= x
  g(x,y)= (x**y)**(1/y)
  ```

  An accuracy sample then, is an evaluation of the degree to which this relation held true.

  The range of argument values for x and y were selected such that x was held constant at a particular value and y was uniformly varied throughout a range of values which avoided overflowing or underflowing the result field. The particular values selected are indicated in the subcolumns entitled x and y.
- *Accuracy Data*: The maximum relative error and standard deviation (root mean square) of the relative error are generally useful and revealing statistics. These statistics for the relative error, (e), are provided in the following subcolumns:

MAX(e) =	MAX(ABS((f(x) - g(x)) / f(x))) where: MAX selects the largest of its arguments and ABS takes the absolute value of its argument.
SD(e) =	SQR((1/N) SUMSQ((f(x) - g(x)) / f(x))) where: SQR takes the square root of its argument and SUMSQ takes the summation of the squares of its arguments over all of the test cases.

Limitations (Subject to Change): The following are limits that apply to the functions performed by this instruction.

The source and *receiver* operands must both be specified as floating-point with the same length (4 bytes for short format or 8 bytes for long format).

Resultant Conditions:

-
- Positive-The algebraic value of the receiver operand is positive.
- Negative-The algebraic value of the receiver operand is negative.
- Zero-The algebraic value of the receiver operand is zero.
- Unordered-The value assigned to the floating-point result is NaN.

Warning: Temporary Level 3 Header

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

0C Computation

- 0C06 Floating-Point Overflow
- 0C07 Floating-Point Underflow
- 0C09 Floating-Point Invalid Operand
- 0C0C Invalid Floating-Point Conversion
- 0C0D Floating-Point Inexact Result
- 0C0E Floating-Point Zero Divide

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

2E02 Security Audit Journal Failure

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compute Time Duration (CTD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0454	Time duration	Time 1	Time 2	Instruction template

Operand 1: Packed decimal variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4: Space pointer.

Bound program access

```
Built-in number for CTD is 102.
CTD (
     time_duration          : address of packed decimal
     time1                  : address
     time2                  : address
     instruction_template   : address
)
```

Description: The time specified by operand 3 is subtracted from the time specified by operand 2 and the resulting *time duration* is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

A negative value will be returned when the *time 1* operand is less than the *time 2* operand.

The following describes the *instruction template*.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Instruction template	Char(*)	
0	0		Instruction template size	UBin(4)
4	4		Operand 1 data definitional attribute template number	UBin(2)
6	6		Operand 2 data definitional attribute template number	UBin(2)
8	8		Operand 3 data definitional attribute template number	UBin(2)
10	A		Reserved (binary 0)	Char(2)
12	C		Operand 2 length	UBin(2)
14	E		Operand 3 length	UBin(2)
16	10		Reserved (binary 0)	Char(26)
42	2A		Data definitional attribute template list	Char(*)
42	2A		Size of the DDAT list	UBin(4)
46	2E		Number of DDATs	UBin(2)
48	30		Reserved (binary 0)	Char(10)
58	3A		DDAT offset	[*] UBin(4)
*	*		Data definitional attribute template	[*] Char(*)
*	*	— End —		

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the *data definitional attribute template list*. For example, the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDAT for operand 1 must be valid for a time duration. The DDATs for operands 2 and 3 must be valid for a time and must be identical. Otherwise, a *template value invalid* (hex 3801) exception will be signaled.

Operand 2 length and **operand 3 length** are specified in number of bytes.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the template. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C16 Data Format Error

0C17 Data Value Error

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Compute Timestamp Duration (CTSD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
043C	Timestamp duration	Timestamp 1	Timestamp 2	Instruction template

Operand 1: Packed decimal variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4: Space pointer.

Bound program access

```
Built-in number for CTSD is 103.
CTSD (
      timestamp_duration     : address of packed decimal
      timestamp1             : address
      timestamp2             : address
      instruction_template   : address
)
```

Description: The timestamp specified by operand 3 is subtracted from the timestamp specified by operand 2 and the resulting *time duration* is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

A negative value will be returned when the *timestamp 1* operand is less than the *timestamp 2* operand.

The following describes the *instruction template*.

Offset				
Dec	Hex	Field Name	Data Type and Length	
0	0	Instruction template	Char(*)	
0	0		Instruction template size	Bin(4)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
4	4		Operand 1 data definitional attribute template number	UBin(2)
6	6		Operand 2 data definitional attribute template number	UBin(2)
8	8		Operand 3 data definitional attribute template number	UBin(2)
10	A		Reserved (binary 0)	Char(2)
12	C		Operand 2 length	UBin(2)
14	E		Operand 3 length	UBin(2)
16	10		Reserved (binary 0)	Char(26)
42	2A		Data definitional attribute template list	Char(*)
42	2A		Size of the DDAT list	UBin(4)
46	2E		Number of DDATs	UBin(2)
48	30		Reserved (binary 0)	Char(10)
58	3A		DDAT offset	[*] UBin(4
*	*		Data definitional attribute template	[*] Char(*
*	*	—- End —-		

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the *data definitional attribute template list*. For example, the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDAT for operand 1 must be valid for a timestamp duration. The DDATs for operands 2 and 3 must be valid for a timestamp and must be identical. Otherwise, a *template value invalid* (hex 3801) exception will be signaled.

Operand 2 length and **operand 3 length** are specified in number of bytes.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the template. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

0C Computation

- 0C15 Date Boundary Overflow
- 0C16 Data Format Error
- 0C17 Data Value Error
- 0C18 Date Boundary Underflow

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Concatenate (CAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10F3	Receiver	Source 1	Source 2

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Description: The character string value of the second source operand is joined to the right end of the character string value of the first source operand. The resulting string value is placed (left-adjusted) in the *receiver* operand.

The length of the operation is equal to the length of the *receiver* operand with the resulting string truncated or is logically padded on the right end accordingly. The pad value for this instruction is hex 40.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1, 2, and 3. The effect of specifying a null substring reference for one source operand is that the other source operand is used as the result of the concatenation. The effect of specifying a null substring reference for both source operands is that the bytes of the receiver are each set with a value of hex 40. The effect of specifying a null substring reference for the *receiver* is that a result is not set regardless of the value of the source operands.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert BSC to Character (CVTBC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
CVTBC 10AF		Receiver	Controls	Source	
CVTBCB 1CAF	Branch options	Receiver	Controls	Source	Branch targets
CVTBCI 18AF	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(3) variable scalar.

Operand 3: Character scalar.

Operand 4-6:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CVTBC is 130.
CVTBC (
       receiver          : address
       receiver_length   : unsigned binary(4)
       controls          : address
       source            : address
       source_length     : unsigned binary(4)
       return_code       : address of signed binary(4)
)
```

The return_code will be set as follows:

Return code	Meaning
-1	Completed Record.
0	Source Exhausted.
1	Truncated Record.

The *receiver*, *controls* and *source* parameters correspond to operands 1, 2 and 3 on the CVTBC operation.

The *receiver_length* and *source_length* parameters contain the length, in bytes, of the receiver and source strings. They are expected to contain values between 1 and 32,767.

The *return_code* parameter is used to provide support for the branch and indicator forms of the CVTBC operation. The user must specify code to process the *return_code* and perform the desired branching or indicator setting.

Description: This instruction converts a string value from the BSC (binary synchronous communications) compressed format to a character string. The operation converts the *source* (operand 3) from the BSC compressed format to character under control of the *controls* (operand 2) and places the result into the *receiver* (operand 1).

The *source* and *receiver* operands must both be character strings.

The *controls* operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 3 bytes in length and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(3)	
0	0		Source offset	Bin(2)
2	2		Record separator	Char(1)
3	3	—- End —-		

The **source offset** specifies the offset where bytes are to be accessed from the *source* operand. If the *source offset* is equal to or greater than the length specified for the *source* operand (it identifies a byte beyond the

end of the *source* operand), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *source offset* is set to specify the offset that indicates how much of the *source* is processed when the instruction ends.

The **record separator**, if specified with a value other than hex 01, contains the value used to separate converted records in the source operand. A value of hex 01 specifies that record separators do not occur in the converted records in the *source*.

Only the first 3 bytes of the *controls* operand are used. Any excess bytes are ignored.

The operation begins by accessing the bytes of the *source* operand located at the offset specified in the *source offset*. This is assumed to be the start of a record. The bytes of the record in the *source* operand are converted into the receiver record according to the following algorithm.

The strings to be built in the *receiver* are contained in the *source* as blank compression entries and strings of consecutive nonblank characters.

The format of the blank compression entries occurring in the source are as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Blank compression entry	Char(2)	
0	0		Interchange group separator	Char(1)
1	1		Count of compressed blanks	Char(1)
2	2	—- End —-		

The **interchange group separator** has a fixed value of hex 1D.

The **count of compressed blanks** provides for describing up to 63 compressed blanks. The count of the number of blanks (up to 63) to be decompressed is formed by subtracting hex 40 from the value of the count field. The count field can vary from a value of hex 41 to hex 7F. If the count field contains a value outside of this range, a *conversion* (hex 0C01) exception is signaled.

Strings of blanks described by blank compression entries in the source are repeated in the *receiver* the number of times specified by the blank compression count.

Nonblank strings in the source are copied into the *receiver* intact with no alteration.

If the receiver record is filled with converted data without encountering the end of the *source* operand, the instruction ends with a resultant condition of *completed record*. This can occur in two ways. If a *record separator* was not specified, the instruction ends when enough bytes have been converted from the *source* to fill the *receiver*. If a *record separator* was specified, the instruction ends when a source byte is encountered with that value prior to or just after filling the receiver record. The *source offset* value locates the byte following the last source record (including the *record separator*) for which conversion was completed. When the *record separator* value is encountered, any remaining bytes in the *receiver* are padded with blanks.

If the end of the *source* operand is encountered (whether or not in conjunction with a *record separator* or the filling of the *receiver)*, the instruction ends with a resultant condition of *source exhausted*. The *source offset* value locates the byte following the last byte of the *source* operand. The remaining bytes in the *receiver* after the converted record are padded with blanks.

If the converted form of a record cannot be completely contained in the *receiver*, the instruction ends with a resultant condition of *truncated record*. The offset value for the *source* locates the byte following the last source byte for which conversion was performed, unless a blank compression entry was being processed.

In this case, the *source offset* is set to locate the byte after the blank compression entry. If the *source* does not contain record separators, this condition can only occur for the case in which a blank compression entry was being converted when the receiver record became full.

Any form of overlap between the operands on this instruction yields unpredictable results in the receiver operand.

Resultant Conditions:

•

- Completed record-The receiver record has been completely filled with converted data from a source record.
- Source exhausted-All of the bytes in the *source* operand have been converted into the receiver operand.
- Truncated record-The receiver record cannot contain all of the converted data from the source record.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C01 Conversion

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Character to BSC (CVTCB)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CVTCB 108F		Receiver	Controls	Source	

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CVTCBB 1C8F	Branch options	Receiver	Controls	Source	Branch targets
CVTCB 188F	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(3) variable scalar.

Operand 3: Character scalar.

Operand 4-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CVTCB is 131.
CVTCB (
       receiver          : address
       receiver_length   : unsigned binary(4)
       controls          : address
       source            : address
       source_length     : unsigned binary(4)
       return_code       : address of signed binary(4)
)
```

The return_code will be set as follows:

Return code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.

The *receiver*, *controls* and *source* parameters correspond to operands 1, 2 and 3 on the CVTCB operation.

The *receiver_length* and *source_length* parameters contain the length, in bytes, of the receiver and source strings. They are expected to contain values between 1 and 32,767.

The *return_code* parameter is used to provide support for the branch and indicator forms of the CVTCB operation. The user must specify code to process the *return_code* and perform the desired branching or indicator setting.

Description: This instruction converts a string value from character to BSC (binary synchronous communications) compressed format. The operation converts the *source* (operand 3) from character to the BSC compressed format under control of the *controls* (operand 2) and places the result into the *receiver* (operand 1).

The *source* and *receiver* operands must both be character strings.

The *controls* operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 3 bytes in length and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(3)	
0	0		Receiver offset	Bin(2)
2	2		Record separator	Char(1)
3	3	—- End —-		

The **receiver offset** specifies the offset where bytes are to be placed into the *receiver* operand. If the *receiver offset* is equal to or greater than the length specified for the *receiver* operand (it identifies a byte beyond the end of the *receiver*), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *receiver offset* is set to specify the offset that indicates how much of the *receiver* has been filled when the instruction ends.

The **record separator**, if specified with a value other than hex 01, contains the value used to separate converted records in the *receiver* operand. A value of hex 01 specifies that record separators are not to be placed into the *receiver* to separate converted records.

Only the first 3 bytes of the *controls* operand are used. Any excess bytes are ignored.

The *source* operand is assumed to be one record. The bytes of the record in the *source* operand are converted into the *receiver* operand at the location specified in the *receiver offset* according to the following algorithm.

The bytes of the source record are interrogated to identify the strings of consecutive blank (hex 40) characters and the strings of consecutive nonblank characters which occur in the source record. Only three or more blank characters are treated as a blank string for purposes of conversion into the *receiver*.

As the blank and nonblank strings are encountered in the *source*, they are packaged into the *receiver*.

Blank strings are reflected in the *receiver* as one or more blank compression entries. The format of the blank compression entries built into the receiver are as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Blank compression entry	Char(2)	
0	0		Interchange group separator	Char(1)
1	1		Count of compressed blanks	Char(1)
2	2	—- End —-		

The **interchange group separator** has a fixed value of hex 1D.

The **count of compressed blanks** provides for compressing up to 63 blanks. The value of the count field is formed by adding hex 40 to the actual number of blanks (up to 63) to be compressed. The count field can vary from a value of hex 43 to hex 7F.

Nonblank strings are copied into the receiver intact with no alteration or additional control information.

When the end of the source record is encountered, the *record separator* value if specified is placed into the *receiver* and the instruction ends with a resultant condition of *source exhausted*. The *receiver offset* value locates the byte following the converted record in the *receiver*. The value of the remaining bytes in the *receiver* after the converted record is unpredictable.

If the converted form of a record cannot be completely contained in the *receiver* (including the *record separator* if specified), the instruction ends with a resultant condition of *receiver overrun*. The *receiver offset* remains unchanged. The remaining bytes in the *receiver*, starting with the byte located by the *receiver offset*, are unpredictable.

Any form of overlap between the operands on this instruction yields unpredictable results in the *receiver* operand.

Resultant Conditions:

•

- Source exhausted-All of the bytes in the *source* operand have been converted into the *receiver* operand.
- Receiver overrun-An overrun condition in the *receiver* operand was detected before all of the bytes in the *source* operand were processed.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Character to Hex (CVTCH)

Op Code (Hex)	Operand 1	Operand 2
1082	Receiver	Source

Operand 1: Character variable scalar.

Operand 2: Character variable scalar.

Description: Each character (8-bit value) of the string value in the *source* operand is converted to a hex digit (4-bit value) and placed in the *receiver* operand. The *source* operand characters must relate to valid hex digits or a *conversion* (hex 0C01) exception is signaled.

Characters	**Hex Digits**
Hex F0-hex F9	Hex 0-hex 9
Hex C1-hex C6	Hex A-hex F

The operation begins with the two operands left-adjusted and proceeds left to right until all the hex digits of the *receiver operand* have been filled. If the *source* operand is too small, it is logically padded on the right with zero characters (hex F0). If the *source* operand is too large, a *length conformance* (hex 0C08) exception or an *invalid operand length* (hex 2A0A) exception is signaled.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the *source* is that the bytes of the *receiver* are each set with a value of hex 00. The effect of specifying a null substring reference for the *receiver* is that no result is set.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C01 Conversion

0C08 Length Conformance

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Character to MRJE (CVTCM)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CVTCM 108B		Receiver	Controls	Source	
CVTCMB 1C8B	Branch options	Receiver	Controls	Source	Branch targets
CVTCMI 188B	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(13) variable scalar.

Operand 3: Character scalar.

Operand 4-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CVTCM is 133.
CVTCM (
       receiver          : address
       receiver_length   : unsigned binary(4)
       controls          : address
       source            : address
       source_length     : unsigned binary(4)
       return_code       : address of signed binary(4)
)
```

The return_code will be set as follows:

Return code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.

The *receiver*, *controls* and *source* parameters correspond to operands 1, 2 and 3 on the CVTCM operation.

The *receiver_length* and *source_length* parameters contain the length, in bytes, of the receiver and source strings. They are expected to contain values between 1 and 32,767.

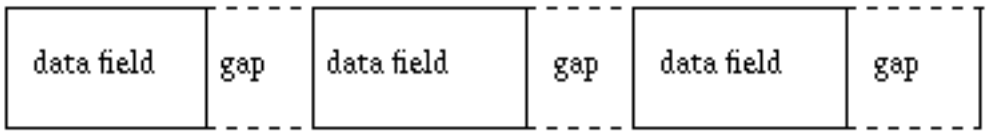
The *return_code* parameter is used to provide support for the branch and indicator forms of the CVTCM operation. The user must specify code to process the *return_code* and perform the desired branching or indicator setting.

Description: This instruction converts a string of characters to MRJE (MULTI-LEAVING remote job entry) compressed format. The operation converts the *source* (operand 3) from character to the MRJE compressed format under control of the *controls* (operand 2) and places the results in the *receiver* (operand 1).

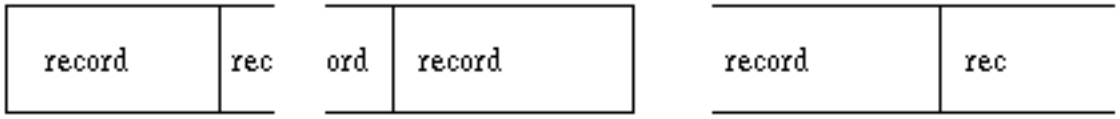
The *source* and *receiver* operands must both be character strings. The *source* operand cannot be specified as either a signed or unsigned immediate value.

The *source* operand can be described through the *controls* operand as being composed of one or more fixed length data fields, which may be separated by fixed length gaps of characters to be ignored during the conversion operation. Additionally, the *controls* operand specifies the amount of data to be processed from the *source* to produce a converted record in the *receiver*. This may be a different value than the length of the data fields in the *source*. The following diagram shows this structure for the *source* operand.

Additional Source Operand Bytes



Data Processed as Source Records



AAC010-00

The *controls* operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 13 bytes in length and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(13)	
0	0		Receiver offset	Bin(2)
2	2		Source offset	Bin(2)
4	4		Algorithm modifier	Char(1)
5	5		Source record length	Char(1)
6	6		Data field length	Bin(2)
8	8		Gap offset	Bin(2)
10	A		Gap length	Bin(2)
12	C		Record control block (RCB) value	Char(1)
13	D	—- End —-		

As input to the instruction, the **source offset** and **receiver offset** fields specify the offsets where bytes of the *source* and *receiver* operands are to be processed. If an offset is equal to or greater than the length specified for the operand it corresponds to (i.e. it identifies a byte beyond the end of the operand), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *source offset* and *receiver offset* fields specify offsets that indicate how much of the operation is complete when the instruction ends.

The **algorithm modifier** has the following valid values:

-
- Hex 00 = Perform full compression.
- Hex 01 = Perform only truncation of trailing blanks.

The **source record length** value specifies the amount of data from the *source* to be processed. If a *source record length* of 0 is specified, a *template value invalid* (hex 3801) exception is signaled.

The **data field length** value specifies the length of the data fields in the *source*. Data fields occurring in the *source* may be separated by gaps of characters, which are to be ignored during the conversion operation. Specification of a *data field length* of 0 indicates that the source operand is one data field. In this case, the *gap length* and *gap offset* values have no meaning and are ignored.

The **gap offset** value specifies the offset to the next gap in the *source*. This value is both input to and output from the instruction. This is relative to the current byte to be processed in the *source* as located by the *source offset* field. No validation is done for this offset. It is assumed to be valid relative to the *source* operand. The *gap offset* value is ignored if the *data field length* is specified with a value of 0.

The **gap length** value specifies the amount of data occurring between data fields in the *source* operand which is to be ignored during the conversion operation. The *gap length* value is ignored if the *data field length* is specified with a value of 0.

The **record control block (RCB) value** field specifies the RCB value that is to precede the converted form of each record in the *receiver*. It can have any value.

Only the first 13 bytes of the *controls* operand are used. Any excess bytes are ignored.

The operation begins by accessing the bytes of the *source* operand at the location specified by the *source offset*. This is assumed to be the start of a source record. Only the bytes of the data fields in the *source* are accessed for conversion purposes. Gaps between data fields are ignored, causing the access of data field bytes to occur as if the data fields were contiguous with one another. Bytes accessed from the source for the source record length are considered a source record for the conversion operation. They are converted into the *receiver* operand at the location specified by the *receiver offset* according to the following algorithm.

The *RCB value* is placed into the first byte of the receiver record.

An SRCB (sub record control byte) value of hex 80 is placed into the second byte of the receiver record.

If the *algorithm modifier* specifies *full compression* (a value of hex 00) then:

The bytes of the source record are interrogated to locate the blank character strings (2 or more consecutive blanks), identical character strings (3 or more consecutive identical characters), and nonidentical character strings occurring in the source. A blank character string occurring at the end of the record is treated as a special case (see following information on trailing blanks).

If the *algorithm modifier* specifies *blank truncation* (a value of hex 01) then:

The bytes of the source record are interrogated to determine if a blank character string exists at the end of the source record. If one exists, it is treated as a string of trailing blanks. All characters prior to it in the record are treated as one string of nonidentical characters.

The strings encountered (blank, identical, or nonidentical) are reflected in the *receiver* by building one or more SCBs (string control bytes) in the *receiver* to describe them.

The format of the SCBs built into the *receiver* is:

•

• SCB format is `o k l jjjjj`

The bit meanings are:

Bit	Value	Meaning
o	0	End of record; the EOR SCB is hex 00.
	1	All other SCBs.

Bit	Value	Meaning
k	0	The string is compressed.
	1	The string is not compressed.
l		For k = 0:
	0	Blanks (hex 40s) have been deleted.
	1	Nonblank characters have been deleted. The next character in the data stream is the specimen character.
		For k = 1:
		This bit is part of the length field for length of uncompressed data.
jjjj		Number of characters that have been deleted if k = 0. The value can be 2-31.
ljjjj		Number of characters to the next SCB (no compression) if k = 1. The value can be 1-63. The uncompressed (nonidentical bytes) follow the SCB in the data stream.

When the end of a source record is encountered, an EOR (end of record) SCB (hex 00) is built into the *receiver*. Trailing blanks in a record including a record of all blanks are represented in the *receiver* by an EOR character. However, a record of all blanks is reflected in the compressed result by an RCB, an SRCB, a compression entry describing an 'unlike string' of one blank character, and an EOR character.

Additionally, the *receiver offset*, the *source offset*, and the *gap offset* are updated in the *controls* operand.

If the end of the *source* operand is not encountered, the operation then continues by reapplying the above algorithm to the next record in the *source* operand.

If the end of the *source* operand is encountered (whether or not in conjunction with a record boundary), the instruction ends with a resultant condition of *source exhausted*. The *source offset* locates the byte following the last source record for which conversion was completed. The *gap offset* value indicates the offset to the next gap relative to the *source offset* value set for this condition. The *gap offset* value has no meaning and is not set when the *data field length* is 0. The *receiver offset* locates the byte following the last fully converted record in the *receiver*. The value of the remaining bytes in the receiver after the last converted record is unpredictable.

If the converted form of a record cannot be completely contained in the *receiver*, the instruction ends with a resultant condition of *receiver overrun*. The *source offset* locates the byte following the last source record for which conversion was completed. The *gap offset* value indicates the offset to the next gap relative to the source offset value set for this condition. The *gap offset* value has no meaning and is not set when the *data field length* is 0. The *receiver offset* locates the byte following the last fully converted record in the *receiver*. The value of the remaining bytes in the *receiver* after the last converted record is unpredictable.

Any form of overlap between the operands of this instruction yields unpredictable results in the *receiver* operand.

Resultant Conditions:

-
- Source exhausted-All complete records in the *source* operand have been converted into the *receiver* operand.
- Receiver overrun-An overrun condition in the *receiver* operand was detected prior to processing all of the bytes in the *source* operand.

If *source exhausted* and *receiver overrun* occur at the same time, the *source exhausted* condition is recognized first. When *source exhausted* is the resultant condition, the *receiver* may also be full. In this case, the *receiver offset* may contain a value equal to the length specified for the *receiver*, and this condition will cause an exception on the next invocation of the instruction. The processing performed for the *source exhausted* condition provides for this case when the instruction is invoked multiple times with the same *controls* operand template. When the *receiver overrun* condition is the resultant condition, the *source* always contains data that can be converted.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Character to Numeric (CVTCN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
1083	Receiver	Source	Attributes

Operand 1: Numeric variable scalar or data-pointer-defined numeric scalar.

Operand 2: Character scalar or data-pointer-defined character scalar.

Operand 3: Character(7) scalar or data-pointer-defined character scalar.

Description: The character scalar specified by operand 2 is treated as though it were a numeric scalar with the attributes specified by operand 3. The character string *source* operand is converted to the numeric forms of the *receiver* operand and moved to the *receiver* operand. The value of operand 2, when viewed in this manner, is converted to the type, length, and precision of the numeric *receiver*, operand 1, following the rules for the Copy Numeric Value (CPYNV) instruction.

The length of operand 2 must be large enough to contain the numeric value described by operand 3. If it is not large enough, a *scalar value invalid* (hex 3203) exception is signaled. If it is larger than needed, its leftmost bytes are used as the value, and the rightmost bytes are ignored.

Normal rules of arithmetic conversion apply except for the following. If operand 2 is interpreted as a zoned decimal value, a value of hex 40 in the rightmost byte referenced in the conversion is treated as a positive sign and a zero digit.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

The format of the *attributes* operand specified by operand 3 is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Scalar attributes	Char(7)		
0	0		Scalar type	Char(1)	
			Hex 00 = Signed binary		
			Hex 01 = Floating-point		
			Hex 02 = Zoned decimal		
			Hex 03 = Packed decimal		
			Hex 0A = Unsigned binary		
1	1		Scalar length	Bin(2)	
			If binary:		
1	1		Length (L) (where L = 2 or 4)		Bits 0-15
			If floating-point:		
1	1		Length (L) (where L = 4 or 8)		Bits 0-15
			If zoned decimal or packed decimal:		
1	1		Fractional digits (F)		Bits 0-7
1	1		Total digits (T)		Bits 8-15
			(where 1 <= T <= 63 and 0 <= F <= T)		
3	3		Reserved (binary 0)	Bin(4)	
7	7	— End —			

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range
- 0604 External Data Object Not Found

08 Argument/Parameter

- 0801 Parameter Reference Violation

0C Computation

- 0C02 Decimal Data
- 0C06 Floating-Point Overflow
- 0C07 Floating-Point Underflow
- 0C09 Floating-Point Invalid Operand
- 0C0A Size
- 0C0C Invalid Floating-Point Conversion
- 0C0D Floating-Point Inexact Result

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Character to SNA (CVTCS)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CVTCS 10CB		Receiver	Controls	Source	
CVTCSB 1CCB	Branch options	Receiver	Controls	Source	Branch targets
CVTCSI 18CB	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(15) variable scalar.

Operand 3: Character scalar.

Operand 4-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CVTCS is 135.
CVTCS (
       receiver          : address
       receiver_length   : unsigned binary(4)
       controls          : address
       source            : address
       source_length     : unsigned binary(4)
       return_code       : address of signed binary(4)
)
```

The return_code will be set as follows:

Return code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.

The *receiver*, *controls* and *source* parameters correspond to operands 1, 2 and 3 on the CVTCS operation.

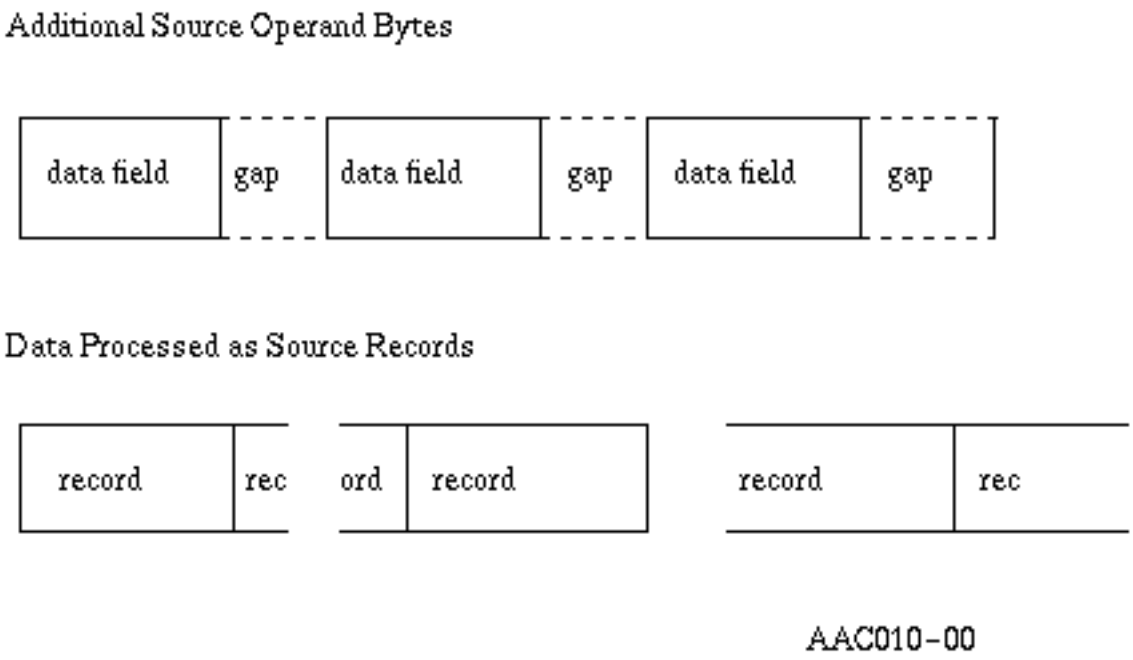
The *receiver_length* and *source_length* parameters contain the length, in bytes, of the receiver and source strings. They are expected to contain values between 1 and 32,767.

The *return_code* parameter is used to provide support for the branch and indicator forms of the CVTCS operation. The user must specify code to process the *return_code* and perform the desired branching or indicator setting.

Description: This instruction converts the *source* (operand 3) from character to SNA (systems network architecture) format under control of the *controls* (operand 2) and places the result into the *receiver* (operand 1).

The *source* and *receiver* operands must both be character strings. The *source* operand may not be specified as an immediate operand.

The source *operand* can be described by the *controls* operand as being one or more fixed-length data fields that may be separated by fixed-length gaps of characters to be ignored during the conversion operation. Additionally, the *controls* operand specifies the amount of data to be processed from the *source* to produce a converted record in the *receiver*. This may be a different value than the length of the data fields in the *source*. The following diagram shows this structure for the *source* operand.



The *controls* operand must be a character scalar that specifies additional information to be used to control the conversion operation. The operand must be at least 15 bytes in length and has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(15)	
0	0		Receiver offset	Bin(2)
2	2		Source offset	Bin(2)
4	4		Algorithm modifier	Char(1)
5	5		Source record length	Char(1)
6	6		Data field length	Bin(2)
8	8		Gap offset	Bin(2)
10	A		Gap length	Bin(2)
12	C		Record separator character	Char(1)
13	D		Prime compression character	Char(1)
14	E		Unconverted source record bytes	Char(1)
15	F	—- End —-		

As input to the instruction, the **source offset** and **receiver offset** fields specify the offsets where the bytes of the *source* and *receiver* operands are to be processed. If an offset is equal to or greater than the length specified for the operand, the offset identifies a byte beyond the end of the operand and a *template value invalid* (hex 3801) exception is signaled. When the *source offset* and the *receiver offset* field are output from the instruction, they specify offsets that indicate how much of the operation is complete when the instruction ends.

The **algorithm modifier** specifies the optional functions to be performed. Any combination of functions can be specified as indicated by the bit meanings in the following chart. At least one of the functions must be specified. If all of the *algorithm modifier* bits are zero, a *template value invalid* (hex 3801) exception is signaled. The *algorithm modifier* bit meanings are:

Bits	Meaning	
0		
	0 =	Do not perform compression.
	1 =	Perform compression.
1-2		
	00 =	Do not use record separators and no blank truncation. Do not perform data transparency conversion.
	01 =	Reserved.
	10 =	Use record separators and perform blank truncation. Do not perform data transparency conversion.
	11 =	Use record separators and perform blank truncation. Perform data transparency conversion.
3		
	0 =	Do not perform record spanning.
	1 =	Perform record spanning. (allowed only when bit 1 = 1)
4-7	(Reserved)	

The **source record length** value specifies the amount of data from the *source* to be processed to produce a converted record in the *receiver*. Specification of a *source record length* of zero results in a *template value invalid* (hex 3801) exception.

The **data field length** value specifies the length of the data fields in the *source*. Data fields occurring in the *source* may be separated by gaps of characters that are to be ignored during the conversion operation. Specification of a *data field length* of zero indicates that the *source* operand is one data field. In this case, the *gap length* and *gap offset* values have no meaning and are ignored.

The **gap offset** value specifies the offset to the next gap in the *source*. This value is both input to and output from the instruction. This is relative to the current byte to be processed in the *source* as located by the *source offset* value. No validation is done for this offset. It is assumed to be valid relative to the *source* operand. The *gap offset* value is ignored if the *data field length* is specified with a value of zero.

The **gap length** value specifies the amount of data that is to be ignored between data fields in the *source* operand during the conversion operation. The *gap length* value is ignored if the *data field length* is zero.

The **record separator character** value specifies the character that precedes the converted form of each record in the *receiver*. It also serves as a delimiter when the previous record is truncating trailing blanks. The Convert SNA to Character instruction recognizes any value that is less than hex 40. The *record separator* value is ignored if *do not use record separators* is specified in the *algorithm modifier*.

The **prime compression character** value specifies the character to be used as the prime compression character when performing compression of the source data to SNA format. It may have any value. The *prime compression character* value is ignored if the *perform compression* function is not specified in the *algorithm modifier*.

The **unconverted source record bytes** value specifies the number of bytes remaining in the current source record that are yet to be converted.

When the *perform record spanning* function is specified in the *algorithm modifier*, the *unconverted source record bytes* field is both input to and output from the instruction. On input, a value of hex 00 means it is the start of a new record and the initial conversion step is yet to be performed. That is, a *record separator character* has not yet been placed in the *receiver*. On input, a nonzero value less than or equal to the *source record length* specifies the number of bytes remaining in the current source record that are yet to be converted into the *receiver*. This value is assumed to be the valid count of unconverted source record bytes relative to the current byte to be processed in the *source* as located by the *source offset* value. As such, it is used to determine the location of the next record boundary in the source operand. This value must be less than or equal to the *source record length* value; otherwise, a *template value invalid* (hex 3801)

exception is signaled. On output this field is set with a value as defined above that describes the number of bytes of the current source record that have not yet been converted.

When the *perform record spanning* function is not specified in the *algorithm modifier*, the *unconverted source record bytes* value is ignored.

Only the first 15 bytes of the *controls* operand are used. Any excess bytes are ignored.

The description of the conversion process is presented as a series of separately performed steps that may be selected in allowable combinations to accomplish the conversion function. It is presented this way to allow for describing these functions separately. However, in the actual execution of the instruction, these functions may be performed in conjunction with one another or separately depending upon which technique is determined to provide the best implementation.

The operation is performed either on a record-by-record basis (record processing) or on a nonrecord basis (string processing). This is determined by the functions selected in the *algorithm modifier*. Specifying the *use record separators* and *perform blank truncation* function indicates **record processing** is to be performed. If this is not specified, in which case *compression* must be specified, it indicates that **string processing** is to be performed.

The operation begins by accessing the bytes of the *source* operand at the location specified by the *source offset*.

When *record processing* is specified, the *source offset* may locate the start of a full or partial record.

When the *perform record spanning* function has not been specified in the *algorithm modifier*, the *source offset* is assumed to locate the start of a record.

When the *perform record spanning* function has been specified in the *algorithm modifier*, the *source offset* is assumed to locate a point at which processing of a possible partially converted record is to be resumed. In this case, the *unconverted source record bytes* value contains the length of the remaining portion of the source record to be converted. The conversion process in this case is started by completing the conversion of the current source record before processing the next full source record.

When *string processing* is specified, the *source offset* locates the start of the source string to be converted.

Only the bytes of the data fields in the source are accessed for conversion purposes. Gaps between data fields are ignored causing the access of data field bytes to occur as if the data fields were contiguous. A string of bytes accessed from the *source* for a length equal to the *source record length* is considered to be a record for the conversion operation.

When during the conversion process, the end of the source operation is encountered, the instruction ends with a resultant condition of *source exhausted*.

When *record processing* is specified in the *algorithm modifier*, this check is performed at the start of conversion for each record. If the source operand does not contain a full record, the *source exhausted* condition is recognized. The instruction is terminated with status in the *controls* operand describing the last completely converted record. For *source exhausted*, partial conversion of a source record is not performed.

When *string processing* is specified in the *algorithm modifier*, then *compression* must be specified and the compression function described below defines the detection of *source exhausted*.

If the converted form of the source cannot be completely contained in the receiver, the instruction ends with a resultant condition of *receiver overrun*. See the description of this condition in the conversion process described below to determine the status of the *controls* operand values and the converted bytes in the receiver for each case.

When *string processing* is specified, the bytes accessed from the source are converted on a string basis into the receiver operand at the location specified by the *receiver offset*. In this case, the *compression* function must be specified and the conversion process proceeds with the compression function defined below.

When *record processing* is specified, the bytes accessed from the source are converted one record at a time into the receiver operand at the location specified by the *receiver offset* performing the functions specified in the *algorithm modifier* in the sequence defined by the following algorithm.

The first function performed is: **trailing blank truncation** A truncated record is built by logically appending the record data to the *record separator* value specified in the *controls* operand and removing all blank characters after the last nonblank character in the record. If a record has no trailing blanks, then no actual truncation takes place. A null record, a record consisting entirely of blanks, will be converted as just the *record separator character* with no other data following it. The truncated record then consists of the *record separator* character followed by the truncated record data, the full record data, or no data from the record.

If either the *data transparency conversion* or the *compression* function is specified in the *algorithm modifier*, the conversion process continues for this record with the next specified function.

If not, the conversion process for this record is completed by placing the truncated record into the *receiver*. If the truncated record cannot be completely contained in the *receiver*, the instruction ends with a resultant condition of *receiver overrun*. When the *perform record spanning* function is specified in the *algorithm modifier*, as much of the truncated record as will fit is placed into the *receiver* and the *controls* operand is updated to describe how much of the source record was successfully converted into the *receiver*. When the *perform record spanning* function is not specified in the *algorithm modifier*, the *controls* operand is updated to describe only the last fully converted record in the *receiver* and the value of the remaining bytes in the *receiver* is unpredictable.

The second function performed is: **data transparency conversion** *Data transparency conversion* is performed if the function is specified in the *algorithm modifier*. This provides for making the data in a record transparent to the Convert SNA to Character instruction in the area of its scanning for record separator values. Transparent data is built by preceding the data with 2 bytes of **transparency control information**. The first byte has a fixed value of hex 35 and is referred to as the TRN (transparency) control character. The second byte is a 1-byte hexadecimal count, a value ranging from 1 to 255 decimal, of the number of bytes of data that follow and is referred to as the **TRN count**. This contains the length of the data and does not include the TRN control information length.

Transparency conversion can be specified only in conjunction with *record processing* and, as such, is performed on the truncated form of the source record. The transparent record is built by preceding the data that follows the *record separator* in the truncated record with the *TRN control information*. The *TRN count* in this case contains the length of just the truncated data for the record and does not include the *record separator* character. For the special case of a null record, no *TRN control information* is placed after the *record separator* character because there is no record data to be made transparent.

If the *compression* function is specified in the *algorithm modifier*, the conversion process continues for this record with the compression function.

If not, the conversion process for this record is completed by placing the transparent record into the receiver. If the transparent record cannot be completely contained in the *receiver*, the instruction ends with a resultant condition of *receiver overrun*.

When the *perform record spanning* function is specified in the *algorithm modifier*, as much of the transparent record as will fit is placed into the *receiver* and the *controls* operand is updated to describe how much of the source record was successfully converted into the *receiver*. The *TRN count* is also adjusted to describe the length of the data successfully converted into the *receiver*; thus, the transparent data for the record is not spanned out of the *receiver*. The remaining bytes of the transparent record, if any, will be processed as a partial source record on the next invocation of the instruction and will be preceded by the appropriate *TRN control information*. For the special case where only 1 to 3 bytes are available at the end of the *receiver*, (not enough room for the *record separator*, the transparency control, and a byte of data) then just the *record separator* is placed in the *receiver* for the record being converted. This can cause up to 2 bytes of unused space at the end of the *receiver*. The value of these unused bytes is unpredictable.

When the *perform record spanning* function is not specified in the *algorithm modifier*, the *controls* operand is updated to describe only the last fully converted record in the *receiver* and the value of the remaining bytes in the *receiver* is unpredictable.

The third function performed is: **compression** *Compression* is performed if the function is specified in the *algorithm modifier*. This provides for reducing the size of strings of duplicate characters in the source data. The source data to be compressed may have assumed a partially converted form at this point as a result of processing for functions specified in the *algorithm modifier*. Compressed data is built by concatenating one or more compression strings together to describe the bytes that make up the converted form of the source data prior to the compression step. The bytes of the converted source data are interrogated to locate the prime compression character strings (two or more consecutive prime compression characters), duplicate character strings (three or more duplicate nonprime characters) and nonduplicate character strings occurring in the *source*.

The character strings encountered (prime, duplicate and nonduplicate) are reflected in the compressed data by building one or more compression strings to describe them. Compression strings are comprised of an SCB (string control byte) possibly followed by one or more bytes of data related to the character string to be described.

The format of an SCB and the description of the data that may follow it are:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	SCB	Char(1)		
0	0		Control		Bits 0-1
			00 =	n nonduplicate characters are between this SCB and the next one; where n is the value of the count field (1-63).	
			01 =	Reserved	
			10 =	This SCB represents n deleted prime compression characters; where n is the value of the count field (2-63). The next byte is the next SCB.	
			11 =	This SCB represents n deleted duplicate characters; where n is the value of the count field (3-63). The next byte contains a specimen of the deleted characters. The byte following the specimen character contains the next SCB.	
0	0		Count		Bits 2-7

Offset Dec	Offset Hex	Field Name	Data Type and Length
			This contains the number of characters that have been deleted for a prime or duplicate string, or the number of characters to the next SCB for a nonduplicate string. A count value of zero cannot be produced.
1	1	—- End —-	

When *record processing* is specified, the compression is performed as follows.

The compression function is performed on just the converted form of the current source record including the *record separator* character. The converted form of the source record prior to the compression step may be a truncated record or a transparent record as described above, depending upon the functions selected in the algorithm modifier. The *record separator* and *TRN control information* is always converted as a nonduplicate compression entry to provide for length adjustment of the *TRN count*, if necessary.

The conversion process for this record is completed by placing the compressed record into the *receiver*. If the compressed record cannot be completely contained in the *receiver*, the instruction ends with a resultant condition of *receiver overrun*.

When the *perform record spanning* function is specified in the *algorithm modifier*, as much of the compressed record as will fit is placed into the *receiver* and the *controls* operand is updated to describe how much of the source record was successfully converted into the *receiver*. The last compression entry placed into the *receiver* may be adjusted if necessary to a length that provides for filling out the *receiver*. This length adjustment applies only to compression entries for nonduplicate strings. Compression entries for duplicate strings are placed in the *receiver* only if they fit with no adjustment. For the special case where data transparency conversion is specified, the transparent data being described is not spanned out of the *receiver*. This is provided for by performing length adjustment on the *TRN count* of a transparent record, which may be included in the compressed data so that it describes only the source data that was successfully converted into the *receiver*. For the special case where only 2 to 5 bytes are available at the end of the *receiver*, not enough room for the compression entry for a nonduplicate string containing the *record separator* and the TRN control, and up to a 2-byte compression entry for some of the transparent data, the nonduplicate compression entry is adjusted to describe only the *record separator*. By doing this, no more than 3 bytes of unused space will remain in the *receiver*. The value of these unused bytes is unpredictable. Unconverted source record bytes, if any, will be processed as a partial source record on the next invocation of the instruction and will be preceded by the appropriate *TRN control information* when performing transparency conversion.

When the *perform record spanning* function is not specified in the *algorithm modifier*, the *controls* operand is updated to describe only the last fully converted record in the *receiver*. The value of the remaining bytes in the receiver is unpredictable.

When *string processing* is specified, the compression is performed as follows.

The compression function is performed on the data for the entire *source* operand on a compression string basis. In this case, the fields in the *controls* operand related to record processing are ignored.

The conversion process for the *source* operand is completed by placing the compressed data into the *receiver*.

When the compressed data cannot be completely contained in the *receiver*, the instruction ends with a resultant condition of *receiver overrun*. As much of the compressed data as will fit is placed into the *receiver* and the *controls* operand is updated to describe how much of the source data was successfully converted into the *receiver*. The last compression entry placed into the *receiver* may be adjusted if

necessary to a length that provides for filling out the *receiver*. This length adjustment applies only to compression entries for nonduplicate strings. Compression entries for duplicate strings are placed in the receiver only if they fit with no adjustment. By doing this, no more than 1 byte of unused space will remain in the *receiver*.

When the compressed data can be completely contained in the *receiver*, the instruction ends with a resultant condition of *source exhausted*. The compressed data is placed into the *receiver* and the *controls* operand is updated to indicate that all of the source data was successfully converted into the *receiver*.

At this point, either conversion of a source record has been completed or conversion has been interrupted due to detection of the *source exhausted* or *receiver overrun* conditions. For record processing, if neither of the above conditions has been detected either during conversion of or at completion of conversion for the current record, the conversion process continues on the next source record with the blank truncation step described above.

At completion of the instruction, the *receiver offset* locates the byte following the last converted byte in the *receiver*. The value of the remaining bytes in the *receiver* after the last converted byte are unpredictable. The *source offset* locates the byte following the last source byte for which conversion was completed. When the *perform record spanning* function is specified in the *algorithm modifier*, the *unconverted source record bytes* field specifies the length of the remaining source record bytes yet to be converted. When the *perform record spanning* function is not specified in the *algorithm modifier*, the *unconverted source record bytes* field has no meaning and is not set. The *gap offset* value indicates the offset to the next gap relative to the source offset value set for this condition. The *gap offset* value has no meaning and is not set when the *data field length* is zero.

Warning: Temporary Level 3 Header

Limitations (Subject to Change)

The following are limits that apply to the functions performed by this instruction.

•

- Any form of overlap between the operands on this instruction yields unpredictable results in the *receiver* operand.
- Substring operand references that allow for a null substring reference (a length value of zero) may not be specified for this instruction.

Resultant Conditions

•

- Source exhausted - All bytes in the *source* operand have been converted into the *receiver* operand.
- Receiver overrun - An overrun condition in the *receiver* operand was detected before all of the bytes in the *source* operand were processed.

Programming Notes:

If the *source* operand does not end on a record boundary, in which case the last record is spanned out of the source, this instruction performs conversion only up to the start of that partial record. In this case, the user of the instruction must move this partial record to combine it with the rest of the record in the source operand to provide for its being processed correctly upon the next invocation of the instruction. If full records are provided, the instruction performs its conversions out to the end of the source operand and no special processing is required.

For the special case of a tie between the *source exhausted* and *receiver overrun* conditions, the *source exhausted* condition is recognized first. That is, when *source exhausted* is the resultant condition, the receiver may also be full. In this case, the *receiver offset* may contain a value equal to the length specified for the *receiver*, which would cause an exception to be detected on the next invocation of the instruction. The processing performed for the *source exhausted* condition should provide for this case if the instruction is to be invoked multiple times with the same *controls* operand template. When the *receiver overrun* condition is the resultant condition, the *source* will always contain data that can be converted.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Date (CVTD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
040F	Result date	Source date	Instruction template

Operand 1: Character variable scalar, packed variable scalar, or zoned variable scalar.

Operand 2: Character scalar, packed scalar, or zoned scalar.

Operand 3: Space pointer.

Bound program access

```
Built-in number for CVTD is 104.
CVTD (
      result_date            : address of  aggregate OR
                               address of  zoned decimal OR
                               address of  packed decimal
      source_date            : address of  aggregate OR
                               address of  zoned decimal OR
                               address of  packed decimal
      instruction_template   : address
)
```

Description: The date specified in operand 2 is converted to another calendar external or internal presentation and placed in operand 1. Operand 3 defines the data definitional attributes for operands 1 and 2.

The following describes the *instruction template*.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Instruction template	Char(*)	
0	0		Instruction template size	Bin(4)
4	4		Operand 1 data definitional attribute template number	UBin(2)
6	6		Operand 2 data definitional attribute template number	UBin(2)
8	8		Reserved (binary 0)	Char(2)
10	A		Operand 1 length	UBin(2)
12	C		Operand 2 length	UBin(2)
14	E		Reserved (binary 0)	Char(2)
16	10		Preferred/Found date format	UBin(2)
18	12		Preferred/Found date separator	Char(1)
19	13		Reserved (binary 0)	Char(23)
42	2A		Data definitional attribute template list	Char(*)
42	2A		Size of the DDAT list	UBin(4
46	2E		Number of DDATs	UBin(2
48	30		Reserved (binary 0)	Char(1
58	3A		DDAT offset	[*] UBi
*	*		Data definitional attribute template	[*] Cha
*	*	—- End —-		

A **data definitional attribute template number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1 and 2.

The DDATs for operands 1 and 2 must be valid for a date. Otherwise, a *template value invalid* (hex 3801) exception will be signaled.

Operand 1 length and **operand 2 length** are specified in number of bytes.

If the *data definitional attribute template numbers* for operands 1 and 2 are the same, only data validation is performed. The validation will check for format and data value correctness.

A format of unknown date, time, or timestamp will indicate that operand 2 will be scanned for a valid format. For a list of formats that can be scanned, see Data Definitional Attribute Template. With an unknown format, the **preferred/found date format** and **preferred/found date separator** can be specified to select an additional non-scanable format. This *preferred format* and *preferred separator* will be used first to find a matching format before scanning operand 2. When the *preferred format* and *preferred separator* have a hex value of zero, only the scan occurs.

When a format of unknown date, time, or timestamp is specified, the *preferred/found date format* and *preferred/found date separator* fields will be set to the format and separator found.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 2.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the templates. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C15 Date Boundary Overflow

0C16 Data Format Error

0C17 Data Value Error

0C18 Date Boundary Underflow

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

- 3202 Scalar Attributes Invalid
- 3203 Scalar Value Invalid

36 Space Management

- 3601 Space Extension/Truncation

38 Template Specification

- 3801 Template Value Invalid

44 Protection Violation

- 4401 Object Domain or Hardware Storage Protection Violation
- 4402 Literal Values Cannot Be Changed

Convert Decimal Form to Floating-Point (CVTDFFP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
107F	Receiver	Decimal exponent	Decimal significand

Operand 1: Floating-point variable scalar.

Operand 2: Packed scalar or zoned scalar (1 to 31 digits).

Operand 3: Packed scalar or zoned scalar (1 to 31 digits).

Description: This instruction converts the decimal form of a floating-point value specified by a *decimal exponent* and a *decimal significand* to binary floating-point format, and places the result in the *receiver* operand. The *decimal exponent* (operand 2) and *decimal significand* (operand 3) are considered to specify a decimal form of a floating-point number. The value of this number is considered to be as follows:

```
Value = S * (10**E)
```

where:

S = The value of the *decimal significand* operand.

E = The value of the *decimal exponent* operand.

* Denotes multiplication.

** Denotes exponentiation.

The *decimal exponent* must be specified as a decimal integer value; no fractional digit positions may be specified in its definition. The *decimal exponent* is a signed integer value specifying a power of 10 which gives the floating-point value its magnitude. A *decimal exponent* value too large or too small to be represented in the *receiver* will result in the signaling of the appropriate *floating-point overflow* (hex 0C06) exception or *floating-point underflow* (hex 0C07) exception.

The *decimal significand* must be specified as a decimal value with a single integer digit position and optional fractional digit positions. The *decimal significand* is a signed decimal value specifying decimal digits which give the floating-point value its precision. The significant digits of the *decimal significand* are considered to start with the leftmost nonzero decimal digit and continue to the right to the end of the *decimal significand* value. Significant digits beyond 7 for a short float *receiver*, and beyond 15 for a long float *receiver* exceed the precision provided for in the binary floating-point receiver. These excess digits do participate in the conversion to provide for uniqueness of the conversion as well as for proper rounding.

The decimal form floating-point value specified by the *decimal exponent* and *decimal significand* operands is converted to a binary floating-point number and rounded to the precision of the result field as follows:

Source values which, in magnitude M, are in the range where (10**31-1) * 10**-31 <= M <= (10**31-1) *10**+31 are converted subject to the normal rounding error defined for the floating-point rounding modes.

Source values which, in magnitude M, are in the range where (10**31-1) * 10**-31 > M > (10**31-1) *10**+31 are converted such that the rounding error incurred on the conversion may exceed that defined above. For round to nearest, this error will not exceed by more than .47 units in the least significant digit position of the result in relation to the error that would be incurred for normal rounding. For the other floating-point rounding modes, this error will not exceed 1.47 units in the least significant digit position of the result.

The converted and rounded value is then assigned to the floating-point *receiver*.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert External Form to Numeric Value (CVTEFN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
1087	Receiver	Source	Mask

Operand 1: Numeric variable scalar or data-pointer-defined numeric scalar.

Operand 2: Character scalar or data-pointer-defined character scalar.

Operand 3: Character(3) scalar, null, or data-pointer-defined character(3) scalar.

Bound program access

```
Built-in number for CVTEFN is 136.
CVTEFN (
  receiver              : address of signed binary OR
                          address of unsigned binary OR
                          address of zoned decimal (1 to 63 digits) OR
                          address of packed decimal (1 to 63 digits)
  receiver_attributes   : address
  source                : address
  source_length         : address of unsigned binary(4)
  mask                  : address
)
```

The *receiver_attributes* is a structure which describes the data attributes of the receiver values. The format of this structure matches that of the third operand on the CVTCN and CVTNC operations.

The *source_length* parameter contains the length, in bytes, of the source string. It is expected to contain a value between 1 and 32,767. If a value less than 1 or greater than 32,767 is specified, a *scalar value invalid* (hex 3203) exception is signaled.

Description: This instruction scans a character string for a valid decimal number in display format, removes the display character, and places the results in the receiver operand. The operation begins by scanning the character string value in the source operand to make sure it is a valid decimal number in display format.

Valid types for the *receiver* are: packed or zoned decimal, signed or unsigned binary.

The character string defined by *source* consists of the following optional entries:

-
- Currency symbol - This value is optional and, if present, must precede any sign and digit values. The valid symbol is determined by *mask*. The currency symbol may be preceded in the field by blank (hex 40) characters.
- Sign symbol - This value is optional and, if present, may precede any digit values (a leading sign) or may follow the digit values (a trailing sign). Valid signs are positive (hex 4E) and negative (hex 60). The sign symbol, if it is a leading sign, may be preceded by blank characters. If the sign symbol is a trailing sign, it must be the rightmost character in the field. Only one sign symbol is allowed.
- Decimal digits - Up to 63 decimal digits may be specified. Valid decimal digits are in the range of hex F0 through hex F9 (0-9). The first decimal digit may be preceded by blank characters (hex 40), but hex 40 values located to the right of the leftmost decimal digit are invalid.

The decimal digits may be divided into two parts by the decimal point symbol: an integer part and a fractional part. Digits to the left of the decimal point are interpreted as integer values. Digits to the right are interpreted as a fractional values. If no decimal point symbol is included, the value is interpreted as an integer value. The valid decimal point symbol is determined by *mask*. If the decimal point symbol precedes the leftmost decimal digit, the digit value is interpreted as a fractional value, and the leftmost decimal digit must be adjacent to the decimal point symbol. If the decimal point follows the rightmost decimal digit, the digit value is interpreted as an integer value, and the rightmost decimal digit must be adjacent to the decimal point.

Decimal digits in the integer portion may optionally have comma symbols separating groups of three digits. The leftmost group may contain one, two, or three decimal digits, and each succeeding group must be preceded by the comma symbol and contain three digits. The comma symbol must be adjacent to a decimal digit on either side. The valid comma symbol is determined by *mask*.

Decimal digits in the fractional portion may not be separated by commas and must be adjacent to one another.

Examples of external formats follow. The following symbols are used.

$	currency symbol
.	decimal point
,	comma
D	digit (hex F0-F9)
	blank (hex 40)
+	positive sign
-	negative sign

Format	Comments
$+DDDD.DD	Currency symbol, leading sign, no comma separators
DD,DDD-	Comma symbol, no fraction, trailing sign
-.DDD	No integer, leading sign
$DDD,DDD-	No fraction, comma symbol, trailing sign
$ + DD.DD	Embedded blanks before digits

If the length of operand 2 is 0, a *scalar value invalid* (hex 3203) exception is signaled.

Mask must indicate a 3-byte character scalar. Byte 1 of the string indicates the byte value that is to be used for the currency symbol. Byte 2 of the string indicates the byte value to be used for the comma symbol. Byte 3 of the string indicates the byte value to be used for the decimal point symbol. Unpredictable results can occur if the same value is used for more than one symbol. If *mask* is null for a non-bound program, the currency symbol (hex 5B), comma (hex 6B), and decimal point (hex 4B) are used.

If the syntax rules are violated, a *conversion* (hex 0C01) exception is signaled. If not, a zoned decimal value is formed from the digits of the display format character string. This number is placed in the receiver operand following the rules of a normal arithmetic conversion.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

A data-pointer-defined *receiver* with 8 byte binary attributes is not supported and will cause a *scalar value invalid* (hex 3203) exception to be signaled.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C01 Conversion

0C0A Size

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Floating-Point to Decimal Form (CVTFPDF)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
CVTFPDF 10BF	Decimal exponent	Decimal significand	Source
CVTFPDFR 12BF	Decimal exponent	Decimal significand	Source

Operand 1: Packed variable scalar or zoned variable scalar (1 to 31 digits).

Operand 2: Packed variable scalar or zoned variable scalar (1 to 31 digits).

Operand 3: Floating-point scalar.

Description: This instruction converts a binary floating-point value to a decimal form of a floating-point value specified by a decimal exponent and a decimal significand, and places the result in the *decimal exponent* and *decimal significand* operands.

The value of this number is considered to be as follows:

```
Value = S * (10**E)
```

where:

S =

The value of the *decimal significand* operand.

E =

The value of the *decimal exponent* operand.

*

Denotes multiplication.

**

Denotes exponentiation.

The decimal exponent must be specified as a decimal integer value. No fractional digit positions are allowed. It must be specified with at least five digit positions. The decimal exponent provides for containing a signed integer value specifying a power of 10 which gives the floating-point value its magnitude.

The decimal significand must be specified as a decimal value with a single integer digit position and optional fractional digit positions. The decimal significand provides for containing a signed decimal value specifying decimal digit which gives the floating-point value its precision. The decimal significand is formed as a normalized value, that is, the leftmost digit position is nonzero for a nonzero source value.

When the *source* contains a representation of a normalized binary floating-point number with decimal significand digits beyond the leftmost 7 digits for a short floating-point *source* or beyond the leftmost 15 digits for a long floating-point *source*, the precision allowed for the binary floating-point *source* is exceeded.

When the *source* contains a representation of a denormalized binary floating-point number, it may provide less precision than the precision of a normalized binary floating-point number, depending on the particular source value. Decimal significand digits exceeding the precision of the *source* are set as a result of the conversion to provide for uniqueness of conversion and are correct, except for rounding errors. These digits are only as precise as the floating-point calculations that produced the source value. The *floating-point inexact result* (hex 0C0D) exception provides a means of detecting loss of precision in floating-point calculations.

The binary floating-point source is converted to a decimal form floating-point value and rounded to the precision of the *decimal significand* operand as follows:

•

• The decimal significand is formed as a normalized value and the decimal exponent is set accordingly.

- For the nonround form of the instruction, the value to be assigned to the decimal significand is adjusted to the precision of the decimal significand, if necessary, according to the current float rounding mode in effect for the thread. For the optional round form of the instruction, the decimal round algorithm is used for the precision adjustment of the decimal significand. The decimal round algorithm overrides the current floating-point rounding mode that is in effect for the thread.
- *Source* values which, in magnitude M, are in the range where (10**31-1) * 10**-31 <= M <= (10**31-1) * 10**+31 are converted subject to the normal rounding error defined for the floating-point rounding modes and the optional round form of the instruction.
- *Source* values which, in magnitude M, are in the range where (10**31-1) * 10**-31 > M > (10**31-1) * 10**+31 are converted such that the rounding error incurred on the conversion may exceed that defined above. For round to nearest and the optional round form of the instruction, this error will not exceed by more than .47 units in the least significant digit position of the result, the error that would be incurred for a correctly rounded result. For the other floating-point rounding modes, this error will not exceed 1.47 units in the least significant digit position of the result.
- If necessary, the decimal exponent value is adjusted to compensate for rounding.
- The converted and rounded value is then assigned to the *decimal exponent* and *decimal significand* operands.

A *size* (hex 0C0A) exception cannot occur on the assignment of the *decimal exponent* or the *decimal significand* values.

Limitations (Subject to Change): The following are limits that apply to the functions performed by this instruction.

-
- The result of the operation is unpredictable for any type of overlap between the *decimal exponent* and *decimal significand* operands.

Warning: Temporary Level 3 Header

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Hex to Character (CVTHC)

Op Code (Hex)	Operand 1	Operand 2
1086	Receiver	Source

Operand 1: Character variable scalar.

Operand 2: Character variable scalar.

Description: Each hex digit (4-bit value) of the string value in the *source* operand is converted to a character (8-bit value) and placed in the *receiver* operand.

Hex Digits	Characters
Hex 0-9 =	Hex F0-F9
Hex A-F =	Hex C1-C6

The operation begins with the two operands left-adjusted and proceeds left to right until all the characters of the *receiver* operand have been filled. If the *source* operand contains fewer hex digits than needed to fill the *receiver*, the excess characters are assigned a value of hex F0. If the *source* operand is too large, a *length conformance* (hex 0C08) exception or an *invalid operand length* (hex 2A0A) exception is signaled.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the *source* is that the bytes of the *receiver* are each set with a value of hex F0. The effect of specifying a null substring reference for the *receiver* is that no result is set.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C08 Length Conformance

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert MRJE to Character (CVTMC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CVTMC 10AB		Receiver	Controls	Source	
CVTMCB 1CAB	Branch options	Receiver	Controls	Source	Branch targets
CVTMCI 18AB	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(6) variable scalar.

Operand 3: Character scalar.

Operand 4-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CVTMC is 132.
CVTMC (
       receiver          : address
       receiver_length   : unsigned binary(4)
       controls          : address
       source            : address
       source_length     : unsigned binary(4)
       return_code       : address of signed binary(4)
)
```

The return_code will be set as follows:

Return code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.

The *receiver*, *controls* and *source* parameters correspond to operands 1, 2 and 3 on the CVTMC operation.

The *receiver_length* and *source_length* parameters contain the length, in bytes, of the receiver and source strings. They are expected to contain values between 1 and 32,767.

The *return_code* parameter is used to provide support for the branch and indicator forms of the CVTMC operation. The user must specify code to process the *return_code* and perform the desired branching or indicator setting.

Description: This instruction converts a character string from the MRJE (MULTI-LEAVING remote job entry) compressed format to character format. The operation converts the *source* (operand 3) from the MRJE compressed format to character format under control of the *controls* (operand 2) and places the results in the *receiver* (operand 1).

The *source* and *receiver* operands must both be character strings. The *source* operand cannot be specified as either a signed or unsigned immediate value.

The *controls* operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 6 bytes in length and have the following format:

Offset				
Dec	Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(6)	
0	0		Receiver offset	Bin(2)
2	2		Source offset	Bin(2)
4	4		Algorithm modifier	Char(1)
5	5		Receiver record length	Char(1)
6	6	—- End —-		

As input to the instruction, the **source offset** and **receiver offset** fields specify the offsets where bytes of the *source* and *receiver* operands are to be processed. If an offset is equal to or greater than the length specified for the operand it corresponds to (it identifies a byte beyond the end of the operand), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *source offset* and *receiver offset* fields specify offsets that indicate how much of the operation is complete when the instruction ends.

The **algorithm modifier** has the following valid values:

•

- Hex 00 = Do not move SRCBs (sub record control bytes) from the *source* into the *receiver*.
- Hex 01 = Move SRCBs from the *source* into the *receiver*.

The **receiver record length** value specifies the record length to be used to convert source records into the *receiver* operand. This length applies to only the string portion of the receiver record and does not include the optional SRCB field. If a *receiver record length* of 0 is specified, a *template value invalid* (hex 3801) exception is signaled.

Only the first 6 bytes of the *controls* operand are used. Any excess bytes are ignored.

The operation begins by accessing the bytes of the *source* operand at the location specified by the *source offset*. This is assumed to be the start of a record. The bytes of the records in the *source* operand are converted into the *receiver* operand at the location specified by the *receiver offset* according to the following algorithm.

The first byte of the source record is considered to be an RCB (record control byte) that is to be ignored during conversion.

The second byte of the source record is considered to be an SRCB. If an *algorithm modifier* of value hex 00 was specified, the SRCB is ignored. If an *algorithm modifier* of value hex 01 was specified, the SRCB is copied into the *receiver*.

The strings to be built in the receiver record are described in the *source* after the SRCB by one or more SCBs (string control bytes).

The format of the SCBs in the *source* are as follows:

```
o k l jjjjj
```

The bit meanings are:

Bit	Value	Meaning
o	0	End of record;the EOR SCB is hex 00.
	1	All other SCBs.
k	0	The string is compressed.
	1	The string is not compressed.
l		For k = 0:
	0	Blanks (hex 40s) have been deleted.

Bit	Value	Meaning
	1	Nonblank characters have been deleted. The next character in the data stream is the specimen character. For k = 1: This bit is part of the length field for length of uncompressed data.
jjjjj		Number of characters that have been deleted if k = 0. The value can be 1-31.
ljjjjj		Number of characters to the next SCB (no compression) if k=1. The value can be 1-63. The uncompressed (nonidentical bytes) follow the SCB in the data stream.

A length of 0 encountered in an SCB results in the signaling of a *conversion* (hex 0C01) exception.

Strings of blanks or nonblank identical characters described in the source record are repeated in the *receiver* the number of times indicated by the SCB count value.

Strings of nonidentical characters described in the source record are moved into the *receiver* for the length indicated by the SCB count value.

When an EOR (end of record) SCB (hex 00) is encountered in the *source*, the *receiver* is padded with blanks out to the end of the current record.

If the converted form of a source record is larger than the *receiver record length*, the instruction is terminated by signaling a *length conformance* (hex 0C08) exception.

If the end of the *source* operand is not encountered, the operation then continues by reapplying the above algorithm to the next record in the *source* operand.

If the end of the *source* operand is encountered (whether or not in conjunction with a record boundary, EOR SCB in the source), the instruction ends with a resultant condition of *source exhausted*. The *receiver offset* locates the byte following the last fully converted record in the *receiver*. The *source offset* locates the byte following the last source record for which conversion is complete. The value of the remaining bytes in the *receiver* after the last converted record are unpredictable.

If the converted form of a record cannot be completely contained in the *receiver*, the instruction ends with a resultant condition of *receiver overrun*. The *receiver offset* locates the byte following the last fully converted record in the receiver. The *source offset* locates the byte following the last source record for which conversion is complete. The value of the remaining bytes in the *receiver* after the last converted record is unpredictable.

If the *source exhausted* and the *receiver overrun* conditions occur at the same time, the *source exhausted* condition is recognized first. In this case, the *receiver offset* may contain a value equal to the length specified for the receiver which causes an exception to be signaled on the next invocation of the instruction. The processing performed for the *source exhausted* condition provides for this case if the instruction is invoked multiple times with the same *controls* operand template. When the *receiver overrun* condition is the resultant condition, the source always contains data that can be converted.

Warning: Temporary Level 3 Header

Limitations (Subject to Change)

The following are limits that apply to the functions performed by this instruction.

•

- Any form of overlap between the operands on this instruction yields unpredictable results in the *receiver* operand.

Resultant Conditions

•

- Source exhausted - All full records in the *source* operand have been converted into the *receiver* operand.
- Receiver overrun - An overrun condition in the *receiver* operand was detected prior to processing all of the bytes in the *source* operand.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C01 Conversion

0C08 Length Conformance

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Numeric to Character (CVTNC)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10A3	Receiver	Source	Attributes

Operand 1: Character variable scalar or data-pointer-defined character scalar.

Operand 2: Numeric scalar or data-pointer-defined numeric scalar.

Operand 3: Character(7) scalar or data-pointer-defined character(7) scalar.

Description: The source numeric value (operand 2) is converted and copied to the receiver character string (operand 1). The *receiver* operand is treated as though it had the *attributes* supplied by operand 3. Operand 1, when viewed in this manner, receives the numeric value of operand 2 following the rules of the Copy Numeric Value (CPYNV) instruction.

The format of operand 3 is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Scalar attributes	Char(7)	
0	0		Scalar type	Char(1)
			Hex 00 = Signed binary	
			Hex 01 = Floating-point	
			Hex 02 = Zoned decimal	
			Hex 03 = Packed decimal	
			Hex 0A = Unsigned binary	
1	1		Scalar length	Bin(2)
			If binary:	
1	1		Length (L) (where L = 2 or 4)	Bits 0
			If floating-point:	
1	1		Length (where L = 4 or 8)	Bits 0
			If zoned decimal or packed decimal:	
1	1		Fractional digits (F)	Bits 0
1	1		Total digits (T)	Bits 8
			(where 1 <= T <= 63 and 0 <= F <= T)	
3	3		Reserved (binary 0)	Bin(4)
7	7	—- End —-		

The byte length of operand 1 must be large enough to contain the numeric value described by operand 3. If it is not large enough, a *scalar value invalid* (hex 3203) exception is signaled. If it is larger than needed, the numeric value is placed in the leftmost bytes and the unneeded rightmost bytes are unchanged by the instruction.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert SNA to Character (CVTSC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
CVTSC 10DB		Receiver	Controls	Source	
CVTSCB 1CDB	Branch options	Receiver	Controls	Source	Branch targets
CVTSCI 18DB	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(14) variable scalar.

Operand 3: Character scalar.

Operand 4-6:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for CVTSC is 134.
CVTSC (
       receiver          : address
       receiver_length   : unsigned binary(4)
       controls          : address
       source            : address
       source_length     : unsigned binary(4)
       return_code       : address of signed binary(4)
)
```

The return_code will be set as follows:

Return code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.
1	Escape Code Encountered.

The *receiver*, *controls* and *source* parameters correspond to operands 1, 2 and 3 on the CVTSC operation.

The *receiver_length* and *source_length* parameters contain the length, in bytes, of the receiver and source strings. They are expected to contain values between 1 and 32,767.

The *return_code* parameter is used to provide support for the branch and indicator forms of the CVTSC operation. The user must specify code to process the *return_code* and perform the desired branching or indicator setting.

Description: This instruction converts a string value from SNA (systems network architecture) format to character. The operation converts the *source* (operand 3) from SNA format to character under control of the *controls* (operand 2) and places the result into the *receiver* (operand 1).

The *source* and *receiver* operands must both be character strings. The *source* operand may not be specified as an immediate operand.

The *controls* operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 14 bytes in length and have the following format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Controls operand base template	Char(14)	
0	0		Receiver offset	Bin(2)
2	2		Source offset	Bin(2)
4	4		Algorithm modifier	Char(1)
5	5		Receiver record length	Char(1)
6	6		Record separator	Char(1)
7	7		Prime compression	Char(1)
8	8		Unconverted receiver record bytes	Char(1)
9	9		Conversion status	Char(2)

Offset Dec	Hex	Field Name	Data Type and Length	
11	B		Unconverted transparency string bytes	Char(1)
12	C		Offset into template to translate table	Bin(2)
14	E	Controls operand optional template extension	Char(64)	
14	E		Record separator translate table	Char(64)
78	4E	—- End —-		

Upon input to the instruction, the **source offset** and **receiver offset** fields specify the offsets where bytes of the *source* and *receiver* operands are to be processed. If an offset is equal to or greater than the length specified for the operand it corresponds to (it identifies a byte beyond the end of the operand), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *source offset* and *receiver offset* are set to specify offsets that indicate how much of the operation is complete when the instruction ends.

The **algorithm modifier** specifies the optional functions to be performed. Any combination of functions not precluded by the bit definitions below is valid except that at least one of the functions must be specified. All *algorithm modifier* bits cannot be zero. Specification of an invalid *algorithm modifier* value results in a *template value invalid* (hex 3801) exception. The meaning of the bits in the *algorithm modifier* is the following:

Bits	Meaning	
0		
	0 =	Do not perform decompression. Interpret a source character value of hex 00 as null.
	1 =	Perform decompression. Interpret a source character value of hex 00 as a record separator.
1-2		
	00 =	No record separators in source, no blank padding. Do not perform data transparency conversion.
	01 =	Reserved.
	10 =	Record separators in source, perform blank padding. Do not perform data transparency conversion.
	11 =	Record separators in source, perform blank padding. Perform data transparency conversion.
3-4		
	00 =	Do not put record separators into receiver.
	01 =	Move record separators from source to receiver (allowed only when bit 1 = 1)
	10 =	Translate record separators from source to receiver (allowed only when bit 1 = 1)
	11 =	Move record separator from controls to receiver.
5-7	Reserved	

The **receiver record length** value specifies the record length to be used to convert source records into the *receiver* operand. This length applies only to the data portion of the receiver record and does not include the optional record separator. Specification of a *receiver record length* of zero results in a *template value invalid* (hex 3801) exception. The *receiver record length* value is ignored if no *record separator processing* is requested in the *algorithm modifier*.

The **record separator value** specifies the character that is to precede the converted form of each record in the *receiver*. The *record separator character* specified in the *controls* operand is used only for the case where the *move record separator from controls to receiver* function is specified in the *algorithm modifier* or where a missing record separator in the source is detected.

The **prime compression** value specifies the character to be used as the prime compression character when performing decompression of the SNA format source data to character. It may have any value. The *prime compression* value is ignored if the *perform decompression* function is not specified in the *algorithm modifier*.

The **unconverted receiver record bytes** value specifies the number of bytes remaining in the current receiver record that are yet to be set with converted bytes from the *source*.

When *record separator processing* is specified in the *algorithm modifier*, this value is both input to and output from the instruction. On input, a value of hex 00 means it is the start of processing for a new record, and the initial conversion step is yet to be performed. This indicates that for the case where a function for putting record separators into the *receiver* is specified in the *algorithm modifier*, a *record separator character* has yet to be placed in the *receiver*. On input, a nonzero value less than or equal to the *receiver record length* specifies the number of bytes remaining in the current receiver record that are yet to be set with converted bytes from the source. This value is assumed to be the valid count of unconverted receiver record bytes relative to the current byte to be processed in the receiver as located by the *receiver offset* field. As such, it is used to determine the location of the next record boundary in the *receiver* operand. This value must be less than or equal to the *receiver record length* value; otherwise, a *template value invalid* (hex 3801) exception is signaled. On output, this field is set with a value as defined above which describes the number of bytes of the current receiver record not yet containing converted data.

When *record separator processing* is not specified in the *algorithm modifier*, this value is ignored.

The **conversion status** field specifies status information for the operation to be performed. The meaning of the bits in the conversion status is the following:

Bits	Meaning
0	
	0 = No transparency string active.
	1 = Transparency string active. Unconverted transparency string bytes value contains the remaining string length.
1-15	Reserved

This field is both input to and output from the instruction. It provides for checkpointing the conversion status over successive executions of the instruction.

If the *conversion status* indicates *transparency string active*, but the *algorithm modifier* does not *specify perform data transparency conversion*, a *template value invalid* (hex 3801) exception is signaled.

The **unconverted transparency string bytes** field specifies the number of bytes remaining to be converted for a partially processed transparency string in the *source*.

When *perform data transparency conversion* is specified in the *algorithm modifier*, the *unconverted transparency string bytes* field can be both input to and output from the instruction.

On input, when the no *transparency string active* status is specified in the *conversion status*, this value is ignored.

On input, when *transparency string active* status is specified in the *conversion status*, this value contains a count for the remaining bytes to be converted for a transparency string in the *source*. A value of hex 00 means the count field for a transparency string is the first byte of data to be processed from the *source* operand. A value of hex 01 through hex FF specifies the count of the remaining bytes to be converted for

a transparency string. This value is assumed to be the valid count of unconverted transparency string bytes relative to the current byte to be processed in the source as located by the *source offset* field.

On output, this value is set if necessary along with the *transparency string active* status to describe a partially converted transparency string. A value of hex 00 will be set if the count field is the next byte to be processed for a transparency string. A value of hex 01 through hex FF specifying the number of remaining bytes to be converted for a transparency string, will be set if the count field has already been processed.

When *do not perform data transparency conversion* is specified in the *algorithm modifier*, the *unconverted transparency string bytes* value is ignored.

The **offset into template to translate table** value specifies the offset from the beginning of the template to the *record separator translate table*. This value is ignored unless the *translate record separators from source to receiver* function is specified in the *algorithm modifier*.

The **record separator translate table** value specifies the translate table to be used in translating record separators specified in the source to the record separator value to be placed into the receiver. It is assumed to be 64 bytes in length, providing for translation of record separator values from hex 00 to hex 3F. This translate table is used only when the *translate record separators from source to receiver* function is specified in the *algorithm modifier*. See the record separator conversion function under the conversion process described below for more detail on the usage of the translate table.

Only the first 14 bytes of the controls operand base template and the optional 64-byte extension area specified for the record separator translate table are used. Any excess bytes are ignored.

The description of the conversion process is presented as a series of separately performed steps, which may be selected in allowable combinations to accomplish the conversion function. It is presented this way to allow for describing these functions separately. However, in the actual execution of the instruction, these functions may be performed in conjunction with one another or separately, depending upon which technique is determined to provide the best implementation.

The operation is performed either on a record-by-record basis, record processing, or on a nonrecord basis, string processing. This is determined by the functions selected in the *algorithm modifier*. Specifying the *record separators in source*, *perform blank padding* or *move record separator from controls to receiver* indicates **record processing** is to be performed. If neither of these functions is specified, in which case *decompression* must be specified, it indicates that **string processing** is to be performed.

The operation begins by accessing the bytes of the *source* operand at the location specified by the *source offset*.

When *record processing* is specified, the *source offset* may locate a point at which processing of a partially converted record is to be resumed or processing for a full record is to be started. The *unconverted receiver record bytes* field indicates whether conversion processing is to be started with a partial or a full record. Additionally, the *transparency string active* indicator in the *conversion status* field indicates whether conversion of a transparency string is active for the case of resumption of processing for a partially converted record. The conversion process is started by completing the conversion of a partial source record if necessary before processing the first full source record.

When *string processing* is specified, the *source offset* is assumed to locate the start of a compression entry.

When during the conversion process the end of the *receiver* operand is encountered, the instruction ends with a *resultant condition* or *receiver overrun*.

When *record processing* is specified in the *algorithm modifier*, this check is performed at the start of conversion for each record. A *source exhausted* condition would be detected before a *receiver overrun*

condition if there is no source data to convert. If the *receiver* operand does not have room for a full record, the *receiver overrun* condition is recognized. The instruction is terminated with status in the controls operand describing the last completely converted record. For *receiver overrun*, partial conversion of a source record is not performed.

When *string processing* is specified in the *algorithm modifier*, then decompression must be specified and the decompression function described below defines the detection of *receiver overrun*.

When during the conversion process the end of the *source* operand is encountered, the instruction ends with a resultant condition of *source exhausted*. See the description of this condition in the conversion process described below to determine the status of the controls operand values and the converted bytes in the *receiver* for each case.

When *string processing* is specified, the bytes accessed from the source are converted on a string basis into the *receiver* operand at the location specified by the *receiver offset*. In this case, the *decompression* function must be specified and the conversion process is accomplished with just the decompression function defined below.

When *record processing* is specified, the bytes accessed from the *source* are converted one record at a time into the *receiver* operand at the location specified by the *receiver offset* performing the functions specified in the *algorithm modifier* in the sequence defined by the following algorithm.

Record separator conversion is performed as requested in the *algorithm modifier* during the initial record separator processing performed as each record is being converted. This provides for controlling the setting of the *record separator* value in the *receiver*.

When the *record separators in source* option is specified, the following algorithm is used to locate them. A record separator is recognized in the *source* when a character value less than hex 40 is encountered. When *do not perform decompression* is specified, a source character value of hex 00 is recognized as a null value rather than as a record separator. In this case, the processing of the current record continues with the next source byte and the *receiver* is not updated. When *perform data transparency conversion* is specified, a character value of hex 35 is recognized as the start of a transparency string rather than as a record separator.

If the *do not put record separators into the receiver* function is specified, the record separator, if any, from the source record being processed is removed from the converted form of the source record and will not be placed in the *receiver*.

If the *move record separators from the source to the receiver* function is specified, the record separator from the source record being processed is left as is in the converted form of the source record and will be placed in the *receiver*.

If the *translate record separators from the source to the receiver* function is specified, the record separator from the source record being processed is translated using the specified translate table, replaced with its translated value in the converted form of the source record, and placed in the *receiver*. The translation is performed as in the translate instruction (XLATE) with the *record separator* value serving as the source byte to be translated. It is used as an index into the specified translate table to select the byte in the translate table that contains the value to which the record separator is to be set. If the selected translate table byte is equal to hex FF, it is recognized as an escape code. The instruction ends with a resultant condition of *escape code* encountered, and the controls operand is set to describe the *conversion status* as of the processing completed just prior to the conversion step for the record separator. If the selected translate table byte is not equal to hex FF, the record separator in the converted form of the record is set to its value.

If the *move record separator from controls to receiver* function is specified, the controls *record separator* value is used in the converted form of the source record and will be placed into the receiver.

When the *record separators in source do blank padding* function is requested, an assumed record separator will be used if a record separator is missing in the source data. In this case, the controls *record separator character* is used as the record separator to precede the converted record if record separators are to be placed in the *receiver*. The conversion process continues, bypassing the record separator conversion step that would normally be performed. The condition of a missing record separator is detected when during initial processing for a full record, the first byte of data is not a record separator character.

Decompression is performed if the function is specified in the *algorithm modifier*. This provides for converting strings of duplicate characters in compressed format in the source back to their full size in the *receiver*. Decompression of the source data is accomplished by concatenating together character strings described by the compression strings occurring in the *source*. The *source offset* value is assumed to locate the start of a compression string. Processing of a partial decompressed record is performed if necessary.

The character strings to be built into the *receiver* are described in the *source* by one or more compression strings. Compression strings are comprised of an SCB (string control byte) possibly followed by one or more bytes of data related to the character string to be built into the *receiver*.

The format of an SCB and the description of the data that may follow it is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	SCB	Char(1)		
0	0		Control		Bits 0-1
			00 =	n nonduplicate characters are between this SCB and the next one; where n is the value of the count field (1-63).	
			01 =	Reserved.	
			10 =	This SCB represents n deleted prime compression characters; where n is the value of the count field (1-63). The next byte is the next SCB.	
			11 =	This SCB represents n deleted duplicate characters; where n is the value of the count field (1-63). The next byte contains a specimen of the deleted characters. The byte following the specimen character contains the next SCB.	
0	0		Count		Bits 2-7
			This contains the number of characters that have been deleted for a prime or duplicate string, or the number of characters to the next SCB for a nonduplicate string. A count value of zero is invalid and results in the signaling of a *conversion* (hex 0C01) exception.		
1	1	—- End —-			

Strings of prime compression characters or duplicate characters described in the *source* are repeated in the decompressed character string the number of times indicated by the *SCB count* value.

Strings of nonduplicate characters described in the source record are formed into a decompressed character string for the length indicated by the *SCB count* value.

If the end of the *source* is encountered prior to the end of a compression string, a *conversion* (hex 0C01) exception is signaled.

When *record processing* is specified, decompression is performed one record at a time. In this case, a *conversion* (hex 0C01) exception is signaled if a compression string describes a character string that would span a record boundary in the *receiver*. If the *source* contains record separators, the case of a missing record separator in the *source* is detected as defined under the initial description of the conversion process. *Record separator conversion*, as requested in the *algorithm modifier*, is performed as the initial step in the building of the decompressed record. A record separator to be placed into the *receiver* is in addition to the data to be converted into *receiver* for the length specified in the *receiver record length* field. The decompression of compression strings from the *source* continues until a *record separator character* for the next record is recognized when the *source* contains record separators, or until the decompressed data required to fill the receiver record has been processed or the end of the *source* is encountered whether record separators are in the *source* or not. Transparency strings encountered in the decompressed character string are not scanned for a record separator value. If the end of the *source* is encountered, the data decompressed to that point appended to the optional record separator for this record forms a partial decompressed record. Otherwise, the decompressed character strings appended to the optional record separator for this record form the decompressed record. The conversion process then continues for this record with the next specified function.

When *string processing* is specified, decompression is performed on a compression string basis with no record oriented processing implied. The conversion process for each compression string from the *source* is completed by placing the decompressed character string into the *receiver*. The conversion process continues decompressing compression strings from the source until the end of the *source* or the *receiver* is encountered. When the end of the *source* operand is encountered, the instruction ends with a resultant condition of *source exhausted*. When a character string cannot be completely contained in the *receiver*, the instruction ends with a resultant condition of *receiver overrun*. For either of the above ending conditions, the *controls* operand is updated to describe the status of the conversion operation as of the last completely converted compression entry. Partial conversion of a compression entry is not performed.

Data transparency conversion is performed if *perform data transparency conversion* is specified in the *algorithm modifier*. This provides for correctly identifying record separators in the *source* even if the data for a record contains values that could be interpreted as record separator values. Processing of active transparency strings is performed if necessary.

A nontransparent record is built by appending the nontransparent and transparent data converted from the record to the record separator for the record. The nontransparent record may be produced from either a partial record from the source or a full record from the *source*. This is accomplished by first accessing the record separator for a full record. The case of a missing record separator in the *source* is detected as defined under the initial description of the conversion process. Record separator conversion as requested in the *algorithm modifier* is performed if it has not already been performed by a prior step; the rest of the source record is scanned for values of less than hex 40.

A value greater than or equal to hex 40 is considered nontransparent data and is concatenated onto the record being built as is.

A value equal to hex 35 identifies the start of a transparency string. A transparency string is comprised of 2 bytes of transparency control information followed by the data to be made transparent to scanning for record separators. The first byte has a fixed value of hex 35 and is referred to as the TRN (transparency) control character. The second byte is a 1-byte hexadecimal count, a value remaining from 1 to 255 decimal, of the number of bytes of data that follow and is referred to as the **TRN count**. A *TRN count* of zero is invalid and causes a *conversion* (hex 0C01) exception. This contains the length of the transparent data and does not include the TRN control information length. The transparent data is concatenated to the nontransparent record being built and is not scanned for record separator characters.

A value equal to hex 00 is recognized as the record separator for the next record only when perform decompression is specified in the *algorithm modifier*. In this case, the nontransparent record is complete. When do not perform decompression is specified in the *algorithm modifier*, a value equal to hex 00 is ignored and is not included as part of the nontransparent data built for the current record.

A value less than hex 40 but not equal to hex 35 is considered to be the record separator for the next record, and the forming of the nontransparent record is complete.

The building of the nontransparent record is completed when the length of the data converted into the *receiver* equals the receiver record length if the record separator for the next record is not encountered prior to that point.

If the end of the *source* is encountered prior to completion of building the nontransparent record, the nontransparent record built up to this point is placed in the *receiver* and the instruction ends with a resultant condition of *source exhausted*. The *controls* operand is updated to describe the status for the partially converted record. This includes describing a partially converted transparency string, if necessary, by setting the *active transparency string status* and the *unconverted transparency string bytes* field.

If the building of the nontransparent record is completed prior to encountering the end of the source, the conversion process continues with the blank padding function described below.

Blank padding is performed if the function is specified in the *algorithm modifier*. This provides for expanding out to the size specified by the *receiver record length* the source records for which trailing blanks have been truncated. The padded record may be produced from either a partial record from the *source* or a full record from the *source*.

The record separator for this record is accessed. The case of a missing record separator in the source is detected as defined under the initial description of the conversion process. *Record separator conversion* as requested in the *algorithm modifier*, is performed if it has not already been performed by a prior step.

The nontruncated data, if any, for the record is appended to the optional record separator for the record. The nontruncated data is determined by scanning the source record for the record separator for the next record. This scan is concluded after processing enough data to completely fill the receiver record or upon encountering the record separator for the next record. The data processed prior to concluding the scan is considered the nontruncated data for the record.

The blanks, if any, required to pad the record out to the nontruncated data for the record, concluding the forming of the padded record.

If the end of the source is encountered during the forming of the padded record, the data processed up to that point, appended to the optional record separator for the record, is placed into the receiver and the instruction ends with a resultant condition of *source exhausted*. The controls operand is updated to describe the status of the partially converted record.

If the forming of the padded record is concluded prior to encountering the end of the *source*, the conversion of the record is completed by placing the converted form of the record into the *receiver*.

At this point, either conversion of a source record has been completed or conversion has been interrupted due to detection of the *source exhausted* or *receiver overrun* condition. For *record processing*, if neither of the above conditions has been detected either during conversion of or at completion of conversion for the current record, the conversion process continues on the next source record with the decompression function described above.

At completion of the instruction, the *receiver offset* locates the byte following the last converted byte in the *receiver*. The value of the remaining bytes in the *receiver* after the last converted byte are unpredictable. The *source offset* locates the byte following the last source byte for which conversion was completed.

When record processing is specified, the *unconverted receiver record bytes* field specifies the length of the receiver record bytes not yet containing converted data. When *perform data transparency conversion* is specified in the *algorithm modifier*, the *conversion status* indicates whether conversion of a transparency string was active and the *unconverted transparency string bytes* field specifies the length of the remaining bytes to be processed for an active transparency string.

This instruction does not provide support for compression entries in the source describing data that would span records in the receiver. SNA data from some systems may violate this restriction and as such be incompatible with the instruction. A provision can be made to avoid this incompatibility by performing the conversion of the SNA data through two invocations of this instruction. The first invocation would specify *decompression* with *no record separator processing*. The second invocation would specify *record separator processing* with no *decompression*. This technique provides for separating the decompression step from record separator processing; thus, the incompatibility is avoided.

This instruction can end with the *escape code encountered* condition. In this case, it is expected that the user of the instruction will want to do some special processing for the record separator causing the condition. In order to resume execution of the instruction, the user will have to set the appropriate value for the *record separator* into the *receiver* and update the *controls* operand *source offset* and *receiver offset* fields correctly to provide for restarting processing at the right points in the *receiver* and *source* operands.

For the special case of a tie between the *source exhausted* and *receiver overrun* conditions, the *source exhausted* condition is recognized first. That is, when *source exhausted* is the resultant condition, the receiver may also be full. In this case, the *receiver offset* may contain a value equal to the length specified for the receiver, which would cause an exception to be detected on the next invocation of the instruction. The processing performed for the *source exhausted* condition should provide for this case if the instruction is to be invoked multiple times with the same controls operand template. When the *receiver overrun* condition is the resultant condition, the source will always contain data that can be converted.

This instruction will, in certain cases, ignore what would normally have been interpreted as a record separator value of hex 00. This applies (hex 00 is ignored) for the special case when *do not perform decompression* and *record separators in source* are specified in the *algorithm modifier*. Note that this does not apply when *perform decompression* is specified, or when *do not perform decompression* and *no record separators in source* and *move record separator from controls to receiver* are specified in the *algorithm modifier*.

Warning: Temporary Level 3 Header

Limitations (Subject to Change)

The following are limits that apply to the functions performed by this instruction.

•

- Any form of overlap between the operands on this instruction yields unpredictable results in the *receiver* operand.

Resultant Conditions

•

- Source exhausted-The end of the *source* operand is encountered and no more bytes from the *source* can be converted.
- Receiver overrun-An overrun condition in the *receiver* operand is detected before all of the bytes in the *source* operand have been processed.
- Escape code encountered-A record separator character is encountered in the *source* operand that is to be treated as an escape code.

Authorization Required

•

- None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C01 Conversion

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Time (CVTT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
041F	Result time	Source time	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Space pointer.

Bound program access

```
Built-in number for CVTT is 105.
CVTT (
      result_time            : address
      source_time            : address
      instruction_template   : address
)
```

Description: The time specified in operand 2 is converted to another external or internal presentation and placed in operand 1. Operand 3 defines the data definitional attributes for operands 1 and 2.

The following describes the *instruction template*.

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Instruction template	Char(*)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0		Instruction template size	Bin(4)
4	4		Operand 1 data definitional attribute template number	UBin(2)
6	6		Operand 2 data definitional attribute template number	UBin(2)
8	8		Reserved (binary 0)	Char(2)
10	A		Operand 1 length	UBin(2)
12	C		Operand 2 length	UBin(2)
14	E		Reserved (binary 0)	Char(2)
16	10		Preferred/Found time format	UBin(2)
18	12		Reserved (binary 0)	Char(1)
19	13		Preferred/Found time separator	Char(1)
20	14		Reserved (binary 0)	Char(22)
42	2A		Data definitional attribute template list	Char(*)
42	2A		Size of the DDAT list	UBin(4)
46	2E		Number of DDATs	UBin(2)
48	30		Reserved (binary 0)	Char(10)
58	3A		DDAT offset	[*] UBin(4)
*	*		Data definitional attribute template	[*] Char(*)
*	*	—- End —-		

A **data definitional attribute template number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1 and 2.

The DDATs for operands 1 and 2 must be valid for a time. Otherwise, a *template value invalid* (hex 3801) exception will be signaled.

Operand 1 length and **operand 2 length** are specified in number of bytes.

If the *data definitional attribute template numbers* for operands 1 and 2 are the same, only data validation is performed. The validation will check for format and data value correctness.

A format of unknown date, time, or timestamp will indicate that operand 2 will be scanned for a valid format. For a list of formats that can be scanned, see Data Definitional Attribute Template. With an unknown format, the **preferred/found time format** and **preferred/found time separator** can be specified to select an additional non-scanable format. This *preferred format* and *preferred separator* will be used first to find a matching format before scanning operand 2. When the *preferred format* and *preferred separator* have a hex value of zero, only the scan occurs.

When a format of unknown date, time, or timestamp is specified, the *preferred/found time format* and *preferred/found time separator* fields will be set to the format and separator found.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 2.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many DDAT offsets as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the templates. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C16 Data Format Error

0C17 Data Value Error

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Convert Timestamp (CVTTS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
043F	Result timestamp	Source timestamp	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Space pointer.

Bound program access

```
Built-in number for CVTTS is 106.
CVTTS (
       result_timestamp       : address
       source_timestamp       : address
       instruction_template   : address
)
```

Description: The timestamp specified in operand 2 is converted to another external or internal presentation and placed in operand 1. Operand 3 defines the data definitional attributes for operands 1 and 2.

The following describes the *instruction template*.

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Instruction template	Char(*)		
0	0		Instruction template size	Bin(4)	
4	4		Operand 1 data definitional attribute template number	UBin(2)	
6	6		Operand 2 data definitional attribute template number	UBin(2)	
8	8		Reserved (binary 0)	Char(2)	
10	A		Operand 1 length	UBin(2)	
12	C		Operand 2 length	UBin(2)	
14	E		Reserved (binary 0)	Char(2)	
16	10		Preferred/Found timestamp format	UBin(2)	
18	12		Preferred/Found date separator	Char(1)	
19	13		Preferred/Found time separator	Char(1)	
20	14		Reserved (binary 0)	Char(22)	
42	2A		Data definitional attribute template list	Char(*)	
42	2A			Size of the DDAT list	UBin(4)
46	2E			Number of DDATs	UBin(2)
48	30			Reserved (binary 0)	Char(10)
58	3A			DDAT offset	[*] UBin(4)
*	*			Data definitional attribute template	[*] Char(*)
*	*	— End —			

A **data definitional attribute template number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1 and 2.

The DDATs for operands 1 and 2 must be valid for a timestamp. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length and **operand 2 length** are specified in number of bytes.

If the *data definitional attribute template numbers* for operands 1 and 2 are the same, only data validation is performed. The validation will check for format and data value correctness.

A format of unknown date, time, or timestamp will indicate that operand 2 will be scanned for a valid format. For a list of formats that can be scanned, see Data Definitional Attribute Template. With an unknown format, the **preferred/found timestamp format**, **preferred/found date separator**, and **preferred/found time separator** can be specified to select an additional non-scanable format. The *preferred format* and *preferred separators* will be used first to find a matching format before scanning operand 2. When the *preferred format* and *preferred separators* have a hex value of zero, only the scan occurs.

When a format of unknown date, time, or timestamp is specified, the *preferred/found timestamp format*, *preferred/found date separator*, and *preferred/found time separator* fields will be set to the format and separators found.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 2.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many DDAT offsets as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the templates. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C15 Date Boundary Overflow

0C16 Data Format Error

0C17 Data Value Error

0C18 Date Boundary Underflow

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bits Arithmetic (CPYBTA)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
102C	Receiver	Source	Offset	Length

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character variable scalar or numeric variable scalar.

Operand 3: Signed or unsigned binary immediate.

Operand 4: Signed or unsigned binary immediate.

Description: This instruction copies the signed bit string *source* operand starting at the specified *offset* for a specified *length* right adjusted to the *receiver* and pads on the left with the sign of the bit string source.

The selected bits from the *source* operand are treated as a signed bit string and copied to the *receiver* value.

The *source* operand can be character or numeric. The leftmost bytes of the *source* operand are used in the operation. The *source* operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus 1.

The *offset* operand indicates which bit of the *source* operand is to be copied, with a offset of zero indicating the leftmost bit of the leftmost byte of the *source* operand.

The *length* operand indicates the number of bits that are to be copied.

If the sum of the *offset* plus the *length* exceeds the length of the *source*, an *invalid operand length* (hex 2A0A) exception is signalled.

Warning: Temporary Level 3 Header

Limitations (Subject to Change)

•

- The length of the *receiver* cannot exceed four bytes.
- The *offset* must have a non-negative value.
- The *length* operand must be an immediate value between 1 and 32.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bits Logical (CPYBTL)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
101C	Receiver	Source	Offset	Length

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character variable scalar or numeric variable scalar.

Operand 3: Signed or unsigned binary immediate.

Operand 4: Signed or unsigned binary immediate.

Description: Copies the unsigned bit string *source* operand starting at the specified *offset* for a specified *length* to the *receiver*.

If the *receiver* is shorter than the *length*, the left most bits are removed to make the *source* bit string conform to the length of the *receiver*. No exceptions are generated when truncation occurs.

The selected bits from the *source* operand are treated as an unsigned bit string and copied right adjusted to the *receiver* and padded on the left with binary 0s.

The *source* operand can be character or numeric. The leftmost bytes of the *source* operand are used in the operation. The *source* operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus 1.

The *offset* operand indicates which bit of the *source* operand is to be copied, with an offset of zero indicating the leftmost bit of the leftmost byte of the *source* operand.

The *length* operand indicates the number of bits that are to be copied.

If the sum of the *offset* plus the *length* exceeds the length of the *source*, an *invalid operand length* (hex 2A0A) exception is signaled.

Warning: Temporary Level 3 Header

Limitations (Subject to Change)

•

- The length of the *receiver* cannot exceed four bytes.
- The *offset* must have a non-negative value.
- The *length* operand must be an immediate value between 1 and 32.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bits with Left Logical Shift (CPYBTLLS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
102F	Receiver	Source	Shift control

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character(2) scalar or unsigned binary(2) scalar.

Description: This instruction copies the bit string value of the *source* operand to the bit string defined by the *receiver* operand with a left logical shift of the source bit string value under control of the *shift control* operand.

The operation results in copying the shifted bit string value of the *source* to the bit string of the *receiver* while padding the *receiver* with bit values of 0 and truncating bit values of the *source* as is appropriate for the specific operation.

No indication is given of truncation of bit values from the shifted *source* value. This is true whether the values truncated are 0 or 1.

The operation is performed such that the bit string of the *source* is considered to be extended on the left and right by an unlimited number of bit string positions of value 0. Additionally, a receiver bit string view (window) with the attributes of the *receiver* is considered to overlay this conceptual bit string value of the *source* starting at the leftmost bit position of the original *source* value. A left logical shift of the conceptual bit string value of the *source* is then performed relative to the *receiver* bit string view according to the shift criteria specified in the *shift control* operand. After the shift, the bit string value then contained within the *receiver* bit string view is copied to the *receiver*.

The *source* and the *receiver* can be either character or numeric. Any numeric operands are interpreted as logical character strings. Due to the operation being treated as a character string operation, the *source* operand may not be specified as a signed immediate operand. Additionally, for a *source* operand specified as an unsigned immediate value, only a 1-byte immediate value may be specified.

The *shift control* operand may be specified as an immediate operand, as a character(2) scalar, or as an unsigned binary(2) scalar. It provides an unsigned binary value indicating the number of bit positions for which the left logical shift of the source bit string value is to be performed. A zero value specifies no shift.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bits with Right Arithmetic Shift (CPYBTRAS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
101B	Receiver	Source	Shift control

Operand 1: Character variable or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character(2) scalar or unsigned binary(2) scalar.

Description: The instruction copies the bit string value of the *source* operand to the bit string defined by the *receiver* operand with a right arithmetic shift of the *source* bit string value under control of the *shift control* operand.

The operation results in copying the shifted bit string value of the *source* to the bit string of the *receiver* while padding the *receiver* with bit values of 0 or 1 depending on the high order bit value of the *source*, and truncating bit values of the *source* as is appropriate for the specific operation.

No indication is given of truncation of bit values from the shifted *source* value. This is true whether the values truncated are 0 or 1.

The operation is performed such that the bit string of the *source* is considered a signed numeric binary value, with the value of the sign bit of the *source* conceptually extended on the left an unlimited number of bit string positions, and conceptually extended on the right by an unlimited number of bit string positions of value 0. Additionally, a *receiver* bit string view (window) with the attributes of the *receiver* is considered to overlay this conceptual bit string value of the *source* starting at the leftmost bit position of the original *source* value. A right arithmetic shift of the conceptual bit string value of the *source* is then performed according to the shift criteria specified in the *shift control* operand. No indication is given of truncation of bit values from the shifted conceptual *source* value. This is true whether the values truncated are 0 or 1. After the shift, the bit string value then contained within the *receiver* bit string view is copied to the *receiver*.

Viewing the bit string value of the *source* and the bit string value copied to the *receiver* as signed numeric, the sign of the value copied to the *receiver* will be the same as the sign of the *source*.

Under some circumstances, such as when the *source* and *receiver* have the same length, a right shift of one bit position is equivalent to dividing the signed numeric bit string value of the *source* by 2 with rounding downward, and assigning a signed numeric bit string equivalent to that result to the *receiver*. For example, if the signed numeric view of the *source* bit string is +9, shifting one bit position right yields +4. However if the signed numeric view of the *source* bit string is -9, shifting one bit position right yields -5.

If all the significant bits of the conceptual *source* bit string are shifted out of the field, the resulting conceptual bit string value will be all zero bits for positive numbers, and all one bits for negative numbers.

The *source* and the *receiver* can be either character or numeric. Any numeric operands are interpreted as logical character strings. Due to the operation being treated as a character string operation, the *source* operand may not be specified as a signed immediate operand. Additionally, for a *source* operand specified as an unsigned immediate value, only a 1-byte immediate value may be specified.

The *shift control* operand may be specified as an immediate operand, as a character(2) scalar, or as a unsigned binary(2) scalar. It provides an unsigned binary value indicating the number of bit positions for which the right logical shift of the source bit string value is to be performed. A zero value specifies no shift.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bits with Right Logical Shift (CPYBTRLS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
103F	Receiver	Source	Shift control

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character(2) scalar or unsigned binary(2) scalar.

Description: This instruction copies the bit string value of the *source* operand to the bit string defined by the *receiver* operand with a right logical shift of the source bit string value under control of the *shift control* operand.

The operation results in copying the shifted bit string value of the *source* to the bit string of the *receiver* while padding the *receiver* with bit values of 0 and truncating bit values of the *source* as is appropriate for the specific operation.

No indication is given of truncation of bit values from the shifted source value. This is true whether the values truncated are 0 or 1.

The operation is performed such that the bit string of the *source* is considered to be extended on the left and right by an unlimited number of bit string positions of value 0. Additionally, a *receiver* bit string view (window) with the attributes of the *receiver* is considered to overlay this conceptual bit string value of the *source* starting at the leftmost bit position of the original *source* value. A right logical shift of the conceptual bit string value of the *source* is then performed relative to the *receiver* bit string view according to the shift criteria specified in the *shift control* operand. After the shift, the bit string value then contained within the receiver bit string view is copied to the *receiver*.

The *source* and the *receiver* can be either character or numeric. Any numeric operands are interpreted as logical character strings. Due to the operation being treated as a character string operation, the *source* operand may not be specified as a signed immediate operand. Additionally, for a *source* operand specified as an unsigned immediate value, only a 1-byte immediate value may be specified.

The *shift control* operand may be specified as an immediate operand, as a character(2) scalar, or as a unsigned binary(2) scalar. It provides an unsigned binary value indicating the number of bit positions for which the right logical shift of the source bit string value is to be performed. A zero value specifies no shift.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

- 3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes (CPYBYTES)

Bound program access

```
Built-in number for CPYBYTES is 9.
CPYBYTES (
          target_string   : address of aggregate(*)
          source_string   : address of aggregate(*)
          copy_length     : unsigned binary(4,8) value which specifies the
                            number of bytes to copy
)
```

Description: A copy from the storage specified by *source string* to the storage specified by *target string* is performed. *Copy length* specifies the number of bytes to copy. It is assumed that sufficient storage exists at the locations specified by *source string* and *target string*.

Pointers cannot be copied using this instruction.

Undefined results can occur if the storage locations specified by *target string* and *source string* overlap.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes Left-Adjusted (CPYBLA)

Op Code (Hex)	Operand 1	Operand 2
10B2	Receiver	Source

Operand 1: Character variable scalar, numeric variable scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 2: Character scalar, numeric scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Description: The logical string value of the *source* operand is copied to the logical string value of the *receiver* operand (no padding done). The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the shorter of the two operands. The copying begins with the two operands left-adjusted and proceeds until the shorter operand has been copied.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either operand is that no result is set.

If either operand is a character variable scalar, it may have a length as great as 16,776,191 bytes.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range
- 0604 External Data Object Not Found

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes Left-Adjusted with Pad (CPYBLAP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10B3	Receiver	Source	Pad

Operand 1: Character variable scalar or numeric variable scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 2: Character scalar, numeric scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 3: Character scalar or numeric scalar.

Description: The logical string value of the *source* operand is copied to the logical string value of the *receiver* operand (padded if needed).

The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the *receiver* operand. If the *source* operand is shorter than the *receiver* operand, the *source* operand is copied to the leftmost bytes of the *receiver* operand, and each excess byte of the receiver operand is assigned the single byte value in the *pad* operand. If the *pad* operand is more than 1 byte in length, only its leftmost byte is used. If the *source* operand is longer than the *receiver* operand, the leftmost bytes of the *source* operand (equal in length to the *receiver* operand) are copied to the *receiver* operand.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the *source* is that the bytes of the *receiver* are each set with the single byte value of the *pad* operand. The effect of specifying a null substring reference for the *receiver* is that no result is set.

If either of the first two operands is a character variable scalar, it may have a length as great as 16,776,191.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes Overlap Left-Adjusted (CPYBOLA)

Op Code (Hex)	Operand 1	Operand 2
10BA	Receiver	Source

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character variable scalar or numeric variable scalar.

Description: The logical string value of the *source* operand is copied to the logical string value of the *receiver* operand (no padding done). The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the shorter of the two operands. The copying begins with the two operands left-adjusted and proceeds until the shorter operand has been copied. The excess bytes in the longer operand are not included in the operation.

Predictable results occur even if two operands overlap because the *source* operand is, in effect, first copied to an intermediate result.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either operand is that no result is set.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes Overlap Left-Adjusted with Pad (CPYBOLAP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10BB	Receiver	Source	Pad

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character variable scalar or numeric variable scalar.

Operand 3: Character scalar or numeric scalar.

Description: The logical string value of the *source* operand is copied to the logical string value of the *receiver* operand.

The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the *receiver* operand. If the *source* operand is shorter than the *receiver* operand, the *source* operand is copied to the leftmost bytes of the *receiver* operand and each excess byte of the *receiver* operand is assigned the single byte value in the *pad* operand. If the *pad* operand is more than 1 byte in length, only its leftmost byte is used. If the source operand is longer than the *receiver* operand, the leftmost bytes of the *source* operand (equal in length to the *receiver* operand) are copied to the *receiver* operand.

Predictable results occur even if two operands overlap because the *source* operand is, in effect, first copied to an intermediate result.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the *source* is that the bytes of the *receiver* are each set with the single byte value of the *pad* operand. The effect of specifying a null substring reference for the *receiver* is that no result is set.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes Overlapping (CPYBO)

Bound program access

```
Built-in number for CPYBO is 570.
CPYBO (
       target_string   : address of aggregate(*)
       source_string   : address of aggregate(*)
       copy_length     : unsigned binary(4,8) value which specifies the
                         number of bytes to copy
)
```

Description: A copy from the storage specified by *source string* to the storage specified by *target string* is performed. *Copy length* specifies the number of bytes to copy. It is assumed that sufficient storage exists at the locations specified by *source string* and *target string*.

Pointers cannot be copied using this instruction.

Results are defined if the storage locations specified by *target string* and *source string* overlap. The result is equivalent to copying the *source string* first to a temporary location and then from the temporary location to the *target string*.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes Repeatedly (CPYBREP)

Op Code (Hex)	Operand 1	Operand 2
10BE	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar.

Operand 2: Numeric scalar or character scalar.

Description: The logical string value of the *source* operand is repeatedly copied to the *receiver* operand until the *receiver* is filled. The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The operation begins with the two operands left-adjusted and continues until the *receiver* operand is completely filled. If the *source* operand is shorter than the *receiver*, it is repeatedly copied from left to right (all or in part) until the *receiver* operand is completely filled. If the *source* operand is longer than the *receiver* operand, the leftmost bytes of the *source* operand (equal in length to the *receiver* operand) are copied to the *receiver* operand.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either operand is that no result is set.

If either operand is a character variable scalar, it may have a length as great as 16,776,191.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes Right-Adjusted (CPYBRA)

Op Code (Hex)	Operand 1	Operand 2
10B6	Receiver	Source

Operand 1: Character variable scalar, numeric variable scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 2: Character scalar, numeric scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Description: The logical string value of the *source* operand is copied to the logical string value of the *receiver* operand (no padding done). The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the shorter of the two operands. The rightmost bytes (equal to the length of the shorter of the two operands) of the source operand are copied to the rightmost bytes of the *receiver* operand. The excess bytes in the longer operand are not included in the operation.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either operand is that no result is set.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes Right-Adjusted with Pad (CPYBRAP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10B7	Receiver	Source	Pad

Operand 1: Character variable scalar, numeric variable scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 2: Character scalar, numeric scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 3: Character scalar or numeric scalar.

Description: The logical string value of the *source* operand is copied to the logical string value of the *receiver* operand (padded if needed). The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the *receiver* operand. If the *source* operand is shorter than the *receiver* operand, the *source* operand is copied to the rightmost bytes of *receiver* operand, and each excess byte is assigned the single byte value in the *pad* operand. If the *pad* operand is more than 1 byte in length, only its leftmost byte is used. If the *source* operand is longer than the *receiver* operand, the rightmost bytes of the *source* operand (equal in length to the *receiver* operand) are copied to the *receiver* operand.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the *source* is that the bytes of the *receiver* are each set with the single byte value of the *pad* operand. The effect of specifying a null substring reference for the *receiver* is that no result is set.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes to Bits Arithmetic (CPYBBTA)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
104C	Receiver	Offset	Length	Source

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Signed binary immediate or unsigned binary immediate.

Operand 3: Signed binary immediate or unsigned binary immediate.

Operand 4: Character variable scalar or numeric variable scalar.

Description: This instruction copies a byte string from the *source* operand to a bit string in the *receiver* operand.

The *source* operand is interpreted as a signed binary value and may be sign extended or truncated on the left to fit into the bit string in the *receiver* operand. No indication is given when truncation occurs.

The location of the bit string in the *receiver* operand is specified by the *offset* operand. The value of the *offset* operand specifies the bit offset from the start of the *receiver* operand to the start of the bit string. Thus, an *offset* operand value of 0 specifies that the bit string starts at the leftmost bit position of the *receiver* operand.

The length of the bit string in the *receiver* operand is specified by the *length* operand. The value of the *length* operand specifies the length of the bit string in bits.

Warning: Temporary Level 3 Header

Limitations (Subject to Change)

The following are limits that apply to the functions performed by this instruction.

•

- If the *source* operand and the bit string in the *receiver* operand overlap, the results are unpredictable.
- A *source* operand longer than 4 bytes may not be specified.
- If the *offset* operand is signed binary immediate, a negative value may not be specified.
- A *length* operand with a value less than 1 or greater than 32 may not be specified.
- The bit string specified by the *offset* operand and the *length* operand may not extend outside the *receiver* operand.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes to Bits Logical (CPYBBTL)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
103C	Receiver	Offset	Length	Source

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Signed binary immediate or unsigned binary immediate.

Operand 3: Signed binary immediate or unsigned binary immediate.

Operand 4: Character variable scalar or numeric variable scalar.

Description: This instruction copies a byte string from the *source* operand to a bit string in the *receiver* operand.

The *source* operand is interpreted as an unsigned binary value and may be padded on the left with 0's or truncated on the left to fit into the bit string in the *receiver* operand. No indication is given when truncation occurs.

The location of the bit string in the *receiver* operand is specified by the *offset* operand. The value of the *offset* operand specifies the bit offset from the start of the *receiver* operand to the start of the bit string. Thus, an *offset* operand value of 0 specifies that the bit string starts at the leftmost bit position of the *receiver* operand.

The length of the bit string in the *receiver* operand is specified by the *length* operand. The value of the *length* operand specifies the length of the bit string in bits.

Warning: Temporary Level 3 Header

Limitations (Subject to Change)

The following are limits that apply to the functions performed by this instruction.

•

- If the *source* operand and the bit string in the *receiver* operand overlap, the results are unpredictable.
- A *source* operand longer than 4 bytes may not be specified.
- If the *offset* operand is signed binary immediate, a negative value may not be specified.
- A *length* operand with a value less than 1 or greater than 32 may not be specified.
- The bit string specified by the *offset* operand and the *length* operand may not extend outside the *receiver* operand.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Bytes with Pointers (CPYBWP)

Op Code (Hex)	Operand 1	Operand 2
0132	Receiver	Source

Operand 1: Character variable scalar, space pointer machine object, or pointer data object.

Operand 2: Character variable scalar, space pointer machine object, pointer data object or null.

Bound program access

```
Built-in number for CPYBWP is 14.
CPYBWP (
        receiver   : address
        source     : address
        length     : unsigned binary(4)
)
```

The *receiver* and *source* operands correspond to operands 1 and 2 on the CPYBWP operation.

The *length* operand contains the length, in bytes, of the receiver and source operands. It is expected to contain a value between 1 and 16,776,704.

Description: This instruction copies either the pointer value or the byte string specified for the *source* operand into the *receiver* operand depending upon whether or not a space pointer machine object is specified as one of the operands.

If either operand is a character variable scalar, it can have a length as great as 16,776,191 bytes.

Operations involving space pointer machine objects perform a pointer value copy operation for only space pointer values or the pointer does not exist state. Due to this, a space pointer machine object may only be specified as an operand in conjunction with another pointer or a null second operand. The pointer does not exist state is copied from the source to the receiver pointer without signaling the *pointer does not exist* (hex 2401) exception. *Source* pointer data objects must either be not set or contain a space pointer value when being copied into a *receiver* space pointer machine object. *Receiver* pointer data objects will be set with either the system default pointer does not exist value or the space pointer value from a *source* space pointer machine object.

If the *source* operand is a synchronization pointer, the pointer will be copied to the *receiver*. However, the copied pointer is not useful, because a synchronization pointer is defined to reside at only a single location in memory. Attempting to use a copied synchronization pointer will cause unpredictable results.

Normal pointer alignment checking is performed on a pointer data object specified as an operand in conjunction with a space pointer machine object.

Operations not involving space pointer machine objects, those involving just data objects as operands, perform a byte string copy of the data for the specified operands.

The value of the byte string specified by operand 2 is copied to the byte string specified by operand 1 (no padding done).

The byte string identified by operand 2 can contain the storage forms of both scalars and pointers. Normal pointer alignment checking is not done.

When the Override Program Attributes (OVRPGATR) instruction is not used to override CPYBWP, the only alignment requirement is that the space addressability alignment of the two operands must be to the same position relative to a 16-byte multiple boundary. A *boundary alignment* (hex 0602) exception is signaled if the alignment is incorrect. The pointer attributes of any complete pointers in the *source* are preserved if they can be completely copied into the *receiver*. Partial pointer storage forms are copied into the receiver as scalar data. Scalars in the *source* are copied to the *receiver* as scalars.

When the OVRPGATR instruction is used to override this instruction, the alignment requirement is removed. If the space addressability alignment of the two operands is the same relative to 16-byte multiple boundary, then this instruction will work the same as stated above. If the space addressability

alignment is different, then this instruction will work like a Copy Bytes Left Adjusted (CPYBLA) and the pointer attributes of any complete pointers in the *source* are not preserved in the *receiver*.

If a pointer data object operand contains a data pointer value upon execution of the instruction, the pointer storage form is copied rather than the scalar described by the data pointer value. The character variable scalar reference allowed on either operand cannot be described through a data pointer value.

The length of the operation is equal to the length of the shorter of the two operands. The copying begins with the two operands left-adjusted and proceeds until completion of the shorter operand.

Operand 1 can specify a space pointer machine object only when operand 2 is a space pointer or null.

If operand 2 is null, operand 1 must define a pointer reference. When operand 2 is null, the pointer identified by operand 1 is set to the system default pointer does not exist value.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Extended Characters Left-Adjusted With Pad (CPYECLAP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
1053	Receiver	Source	Pad

Operand 1: Data-pointer-defined character scalar.

Operand 2: Data-pointer-defined character scalar.

Operand 3: Character(3) scalar or null.

Bound program access

```
Built-in number for CPYECLAP is 412.
CPYECLAP (
          receiver_pointer   : address of data pointer
          source_pointer     : address of data pointer
          pad                : address OR
                               null operand
)
```

Description: The extended character string value of the *source* operand is copied to the *receiver* operand.

The operation is performed at the length of the *receiver* operand. If the *source* operand is shorter than the *receiver*, the *source* operand is copied to the leftmost bytes of the *receiver* and the excess bytes of the *receiver* are assigned the appropriate value from the *pad* operand.

The *pad* operand, operand 3, is three bytes in length and has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Pad operand	Char(3)	
0	0		Single byte pad value	Char(1)
1	1		Double byte pad value	Char(2)
3	3	—- End —-		

If the **pad operand** is more than three bytes in length, only its leftmost three bytes are used. Specifying a null *pad operand* results in default pad values of hex 40, for single byte, and hex 4040, for double byte, being used. The **single byte pad value** and the first byte of the **double byte pad value** cannot be either a shift out control character (SO = hex 0E) value or a shift in control character (SI = hex 0F) value. Specification of such an invalid value results in the signaling of the *scalar value invalid* (hex 3203) exception.

Operands 1 and 2 must be specified as data pointers which define either a simple (single byte) character data field or one of the extended (double byte) character data fields.

Support for usage of a data pointer defining an extended character scalar value is limited to this instruction. Usage of such a data pointer defined value on any other instruction is not supported and results in the signaling of the *scalar type invalid* (hex 3201) exception.

For more information on support for extended character data fields, refer to the Set Data Pointer Attributes (SETDPAT) and Materialize Pointer (MATPTR) instructions.

Four data types are supported for data pointer definition of extended (double byte) character fields, OPEN, EITHER, ONLYNS and ONLYS. Except for ONLYNS, the double byte character data must be surrounded by a shift out control character (SO = hex 0E) and a shift in control character (SI = hex 0F).

-
- The ONLYNS field only contains double byte data with no SO, SI delimiters surrounding it.
- The ONLYS field can only contain double byte character data within a SO..SI pair.
- The EITHER field can consist of double byte character or single byte character data but only one type at a time. If double byte character data is present it must be surrounded by an SO..SI pair.
- The OPEN field can consist of a mixture of double byte character and single byte character data. If double byte character data is present it must be surrounded by an SO..SI pair.

Specifying an extended character value which violates the above restrictions results in the signaling of the *invalid extended character data* (hex 0C12) exception.

The valid copy operations which can be specified on this instruction are the following:

Table 1. Valid copy operations for CPYECLAP

	Op1 1			
Op 2	**Onlyns**	**Onlys**	**Open**	**Either**
Onlyns	yes	yes	yes	yes
Onlys	yes	yes	yes	yes
Open	no	no	yes	no

	Op1 1			
Op 2	**Onlyns**	**Onlys**	**Open**	**Either**
Either	no	no	yes	yes

Specifying a copy operation other than the valid operations defined above results in the signaling of the *invalid extended character operation* (hex 0C13) exception.

When the copy operation is for a source of type ONLYNS (no SO/SI delimiters) being copied to a receiver which is not ONLYNS, SO and SI delimiters are implicitly added around the source value as part of the copy operation.

When the source value is longer than can be contained in the receiver, truncation is necessary and the following truncation rules apply:

1. Truncation is on the right (like simple character copy operations).
2. When the string to be truncated is a single byte character string, or an extended character string when the receiver is ONLYNS, bytes beyond those that fit into the receiver are truncated with no further processing needed.
3. When the string to be truncated is an extended character string and the receiver is not ONLYNS, the bytes that fall at the end of the receiver are truncated as follows:
   a. When the last byte that would fit in the receiver is the first byte of an extended character, that byte is truncated and replaced with an SI character.
   b. When the last byte that would fit in the receiver is the second byte of an extended character, both bytes of that extended character are truncated and replaced with a SI character followed by a *single byte pad value*. This type of truncation can only occur when converting to an OPEN field.

When the source value is shorter than that which can be contained in the receiver, padding is necessary. One of three types of padding is performed:

1. Double byte (DB) - the source value is padded on the right with *double byte pad values* out to the length of the receiver.
2. Double byte concatenated with a SI value (DB||SI) - the source double byte value is padded on the right with *double byte pad values* out to the second to last byte of the receiver and an SI delimiter is placed in the last byte of the receiver.
3. Single byte (SB) - the source value is padded on the right with *single byte pad values* out to the length of the receiver.

The type of padding performed is determined by the type of operands involved in the operation:

1. If the receiver is ONLYNS, DB padding is performed.
2. If the receiver is ONLYS, DB||SI padding will be performed.
3. If the receiver is EITHER and the source contained a double byte value, DB||SI padding is performed.
4. If the receiver is EITHER and the source contained a single byte value, SB padding is performed.
5. If the receiver is OPEN, SB padding is performed.

The above padding rules cover all the operand combinations which are allowed on the instruction. A complete understanding of the operand combinations allowed (prior diagram), and the values which can be contained in the different operand types is necessary to appreciate that these rules do cover all the valid combinations.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C12 Invalid Extended Character Data

0C13 Invalid Extended Character Operation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Hex Digit Numeric to Numeric (CPYHEXNN)

Op Code (Hex)	Operand 1	Operand 2
1092	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar.

Operand 2: Numeric scalar or character scalar.

Description: The numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the *source* operand is copied to the numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the *receiver* operand. The operands can be either character strings or numeric. Any numeric operands are interpreted as logical character strings.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

36 Space Management

- 3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Hex Digit Numeric to Zone (CPYHEXNZ)

Op Code (Hex)	Operand 1	Operand 2
1096	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar.

Operand 2: Numeric scalar or character scalar.

Description: The numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the *source* operand is copied to the numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the *receiver* operand. The operands can be either character strings or numeric. Any numeric operands are interpreted as logical character strings.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Hex Digit Zone To Numeric (CPYHEXZN)

Op Code (Hex)	Operand 1	Operand 2
109A	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar.

Operand 2: Numeric scalar or character scalar.

Description: The zone hex digit value (leftmost 4 bits) of the leftmost byte referred to by the *source* operand is copied to the numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the *receiver* operand.

The operands can be either character strings or numeric. Any numeric operands are interpreted as logical character strings.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Hex Digit Zone To Zone (CPYHEXZZ)

Op Code (Hex)	Operand 1	Operand 2
109E	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar.

Operand 2: Numeric scalar or character scalar.

Description: The zone hex digit value (leftmost 4 bits) of the leftmost byte referred to by the *source* operand is copied to the zone hex digit value (leftmost 4 bits) of the leftmost byte referred to by the *receiver* operand.

The operands can be either character strings or numeric. Any numeric operands are interpreted as logical character strings.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Null-Terminated String Constrained (STRNCPYNULL)

Bound program access

```
Built-in number for STRNCPYNULL is 13.
STRNCPYNULL (
             target_string                  : address of aggregate(*)
             null_terminated_source_string  : address of aggregate(*)
             maximum_length                 : unsigned binary(4) value
                                              which specifies the
                                              maximum number of bytes
                                              to copy
) :   space pointer(16) to the target string
```

Description: A copy is performed from the storage specified by *null terminated source string* to the storage specified by *target string*. The copy terminates after a null (ie. zero) character is copied *or* the number of bytes specified by *maximum length* have been copied (which ever comes first). However, if a null character is not copied, one will be appended to the end of the target string. Thus, the maximum number of characters which may be copied is one more than the value of *maximum length.* It is expected that sufficient storage exists at the location specified by *target string.*

Undefined results can occur if the storage locations specified by *target string* and *null terminated source string* overlap.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Null-Terminated String Constrained, Null Padded (STRNCPYNULLPAD)

Bound program access

```
Built-in number for STRNCPYNULLPAD is 12.
STRNCPYNULLPAD (
                target_string                   : address of aggregate(*)
                null_terminated_source_string   : address of aggregate(*)
                maximum_length                  : unsigned binary(4) value
                                                  which specifies the
                                                  maximum number of bytes
                                                  to copy
) :   space pointer(16) to the target string
```

Description: A copy is performed from the storage specified by *null terminated source string* to the storage specified by *target string.* The copy terminates after the number of bytes specified by *maximum length* have been copied. However, if a null (ie. zero) character appears in the *null terminated source string,* no further data is copied from the string. Instead, the *target string* is padded with null characters. It is expected that sufficient storage exists at the location specified by *target string.*

Undefined results can occur if the storage locations specified by *target string* and *null terminated source string* overlap.

Note: If a null character does not appear in the *null terminated source string,* the *target string* will not be a null-terminated string.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Numeric Value (CPYNV)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
CPYNV 1042		Receiver	Source	
CPYNVR 1242		Receiver	Source	
CPYNVB 1C42	Branch options	Receiver	Source	Branch targets
CPYNVBR 1E42	Branch options	Receiver	Source	Branch targets
CPYNVI 1842	Indicator options	Receiver	Source	Indicator targets
CPYNVIR 1A42	Indicator options	Receiver	Source	Indicator targets

Operand 1: Numeric variable scalar or data-pointer-defined numeric scalar.

Operand 2: Numeric scalar or data pointer-defined-numeric scalar.

Operand 3-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for LBCPYNV is 129.
LBCPYNV (
  receiver              : address of  signed binary OR
                          address of  unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits) OR
                          address of  floating point
  receiver_attributes   : address (See SETDPAT for format of attributes)
  source                : address of signed binary OR
                          address of unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits) OR
                          address of floating point
  source_attributes     : address (See SETDPAT for format of attributes)
)

-- OR --

Built-in number for LBCPYNVR is 478.

LBCPYNVR (
  receiver              : address of  signed binary OR
                          address of  unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits)
  receiver_attributes   : address (See SETDPAT for format of attributes)
  source                : address of signed binary OR
                          address of unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits) OR
                          address of floating point
  source_attributes     : address (See SETDPAT for format of attributes)
)
```

Description: The numeric value of the *source* operand is copied to the numeric *receiver* operand.

Both operands must be numeric. If necessary, the *source* operand is converted to the same type as the *receiver* operand before being copied to the *receiver* operand. The *source* value is adjusted to the length of the *receiver* operand, aligned at the assumed decimal point of the *receiver* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations outlined in the Arithmetic Operations. If significant digits are truncated on the left end of the source value, a *size* (hex 0C0A) exception is signaled. When the *receiver* is binary, the *size* (hex 0C0A) exception may be suppressed using program creation options or by changing the *suppress binary size exception attribute* program attribute using the Override Program Attributes (OVRPGATR) instruction.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled or if the *size* (hex 0C0A) exception is suppressed, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Conversions between floating-point integers and integer formats (binary or decimal with no fractional digits) is exact, except when an exception is signaled.

An *invalid floating-point conversion* (hex 0C0C) exception is signaled when an attempt is made to convert from floating-point to binary or decimal and the result would represent infinity or NaN, or nonzero digits would be truncated from the left end of the resultant value.

For the optional round form of the instruction, a floating-point *receiver* operand is invalid.

For a fixed-point operation, if significant digits are truncated from the left end of the *source* value, a *size* (hex 0C0A) exception is signaled.

For a floating-point *receiver*, if the exponent of the resultant value is too large or too small to be represented in the *receiver* field, the *floating-point overflow* (hex 0C06) exception and *floating-point underflow* (hex 0C07) exception are signaled, respectively.

Resultant Conditions:

•

- Positive-The algebraic value of the numeric scalar *receiver* operand is positive.
- Negative-The algebraic value of the numeric scalar *receiver* operand is negative.
- Zero-The algebraic value of the numeric scalar *receiver* operand is zero.
- Unordered-The value assigned a floating-point *receiver* operand is NaN.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Numeric Value (CPYNV)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
CPYNV 1042		Receiver	Source	
CPYNVR 1242		Receiver	Source	
CPYNVB 1C42	Branch options	Receiver	Source	Branch targets
CPYNVBR 1E42	Branch options	Receiver	Source	Branch targets
CPYNVI 1842	Indicator options	Receiver	Source	Indicator targets
CPYNVIR 1A42	Indicator options	Receiver	Source	Indicator targets

Operand 1: Numeric variable scalar or data-pointer-defined numeric scalar.

Operand 2: Numeric scalar or data pointer-defined-numeric scalar.

Operand 3-6:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for LBCPYNV is 129.
LBCPYNV (
  receiver              : address of  signed binary OR
                          address of  unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits) OR
                          address of  floating point
  receiver_attributes   : address (See SETDPAT for format of attributes)
  source                : address of signed binary OR
                          address of unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits) OR
                          address of floating point
  source_attributes     : address (See SETDPAT for format of attributes)
)

-- OR --

Built-in number for LBCPYNVR is 478.

LBCPYNVR (
  receiver              : address of  signed binary OR
                          address of  unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits)
  receiver_attributes   : address (See SETDPAT for format of attributes)
  source                : address of signed binary OR
                          address of unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits) OR
                          address of floating point
  source_attributes     : address (See SETDPAT for format of attributes)
)
```

Description: The numeric value of the *source* operand is copied to the numeric *receiver* operand.

Both operands must be numeric. If necessary, the *source* operand is converted to the same type as the *receiver* operand before being copied to the *receiver* operand. The *source* value is adjusted to the length of the *receiver* operand, aligned at the assumed decimal point of the *receiver* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations outlined in the Arithmetic Operations. If significant digits are truncated on the left end of the source value, a *size* (hex 0C0A) exception is signaled. When the *receiver* is binary, the *size* (hex 0C0A) exception may be suppressed using program creation options or by changing the *suppress binary size exception attribute* program attribute using the Override Program Attributes (OVRPGATR) instruction.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled or if the *size* (hex 0C0A) exception is suppressed, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Conversions between floating-point integers and integer formats (binary or decimal with no fractional digits) is exact, except when an exception is signaled.

An *invalid floating-point conversion* (hex 0C0C) exception is signaled when an attempt is made to convert from floating-point to binary or decimal and the result would represent infinity or NaN, or nonzero digits would be truncated from the left end of the resultant value.

For the optional round form of the instruction, a floating-point *receiver* operand is invalid.

For a fixed-point operation, if significant digits are truncated from the left end of the *source* value, a *size* (hex 0C0A) exception is signaled.

For a floating-point *receiver*, if the exponent of the resultant value is too large or too small to be represented in the *receiver* field, the *floating-point overflow* (hex 0C06) exception and *floating-point underflow* (hex 0C07) exception are signaled, respectively.

Resultant Conditions:

•

- Positive-The algebraic value of the numeric scalar *receiver* operand is positive.
- Negative-The algebraic value of the numeric scalar *receiver* operand is negative.
- Zero-The algebraic value of the numeric scalar *receiver* operand is zero.
- Unordered-The value assigned a floating-point *receiver* operand is NaN.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Copy Numeric Value (CPYNV)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
CPYNV 1042		Receiver	Source	
CPYNVR 1242		Receiver	Source	
CPYNVB 1C42	Branch options	Receiver	Source	Branch targets
CPYNVBR 1E42	Branch options	Receiver	Source	Branch targets
CPYNVI 1842	Indicator options	Receiver	Source	Indicator targets
CPYNVIR 1A42	Indicator options	Receiver	Source	Indicator targets

Operand 1: Numeric variable scalar or data-pointer-defined numeric scalar.

Operand 2: Numeric scalar or data pointer-defined-numeric scalar.

Operand 3-6:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for LBCPYNV is 129.
LBCPYNV (
  receiver              : address of  signed binary OR
                          address of  unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits) OR
                          address of  floating point
  receiver_attributes   : address (See SETDPAT for format of attributes)
  source                : address of signed binary OR
                          address of unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits) OR
                          address of floating point
  source_attributes     : address (See SETDPAT for format of attributes)
)

-- OR --

Built-in number for LBCPYNVR is 478.

LBCPYNVR (
  receiver              : address of  signed binary OR
                          address of  unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits)
  receiver_attributes   : address (See SETDPAT for format of attributes)
  source                : address of signed binary OR
                          address of unsigned binary OR
                          address of  zoned decimal (1 to 63 digits) OR
                          address of  packed decimal (1 to 63 digits) OR
                          address of floating point
  source_attributes     : address (See SETDPAT for format of attributes)
)
```

Description: The numeric value of the *source* operand is copied to the numeric *receiver* operand.

Both operands must be numeric. If necessary, the *source* operand is converted to the same type as the *receiver* operand before being copied to the *receiver* operand. The *source* value is adjusted to the length of the *receiver* operand, aligned at the assumed decimal point of the *receiver* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations outlined in the Arithmetic Operations. If significant digits are truncated on the left end of the source value, a *size* (hex 0C0A) exception is signaled. When the *receiver* is binary, the *size* (hex 0C0A) exception may be suppressed using program creation options or by changing the *suppress binary size exception attribute* program attribute using the Override Program Attributes (OVRPGATR) instruction.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled or if the *size* (hex 0C0A) exception is suppressed, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Conversions between floating-point integers and integer formats (binary or decimal with no fractional digits) is exact, except when an exception is signaled.

An *invalid floating-point conversion* (hex 0C0C) exception is signaled when an attempt is made to convert from floating-point to binary or decimal and the result would represent infinity or NaN, or nonzero digits would be truncated from the left end of the resultant value.

For the optional round form of the instruction, a floating-point *receiver* operand is invalid.

For a fixed-point operation, if significant digits are truncated from the left end of the *source* value, a *size* (hex 0C0A) exception is signaled.

For a floating-point *receiver*, if the exponent of the resultant value is too large or too small to be represented in the *receiver* field, the *floating-point overflow* (hex 0C06) exception and *floating-point underflow* (hex 0C07) exception are signaled, respectively.

Resultant Conditions:

-
- Positive-The algebraic value of the numeric scalar *receiver* operand is positive.
- Negative-The algebraic value of the numeric scalar *receiver* operand is negative.
- Zero-The algebraic value of the numeric scalar *receiver* operand is zero.
- Unordered-The value assigned a floating-point *receiver* operand is NaN.

Warning: Temporary Level 3 Header

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Cosine (COS)

Bound program access
Built-in number for COS is 400. COS (source : floating point(8) value) : floating point(8) value which is the cosine of the source value

Description: The cosine of the numeric value of the *source* operand, whose value is considered to be in radians, is computed and the result is returned.

The result is in the range:

```
-1 <= COS(source) <= 1
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Cosine Hyperbolic (COSH)

Bound program access
Built-in number for COSH is 408. COSH (source : floating point(8) value) : floating point(8) value which is the cosine hyperbolic of the source value

Description: The cosine hyperbolic of the numeric value of the *source* operand is computed and the result (in radians) is returned.

The result is in the range:

```
+1 <= COSH(source) <= +infinity
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Cotangent (COT)

Bound program access
Built-in number for COT is 404. COT (source : floating point(8) value) : floating point(8) value which is the cotangent of the source value

Description: The cotangent of the numeric value of the *source* operand, whose value is considered to be in radians, is computed and the result is returned.

The result is in the range:

```
-infinity <= COT(source) <= +infinity
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Create Activation Group-Based Heap Space (CRTHS)

Op Code (Hex)	Operand 1	Operand 2
03B2	Heap identifier	Creation template

Operand 1: Binary(4) variable scalar.

Operand 2: Space pointer.

Bound program access

```
Built-in number for CRTHS is 112.
CRTHS (
       heap_identifier     : address of signed binary(4) OR
                             address of unsigned binary(4)
       creation_template   : address
)
```

Note: The term ″heap space″ in this instruction refers to an ″activation group-based heap space″.

Description: A heap space is created with the attributes supplied in the heap space *creation template* specified by operand 2. The heap space identifier used to perform allocations and marks against the heap space is returned in operand 1.

The *heap identifier* returned in operand 1 represents the heap space. This identifier is used for the Allocate Activation Group-Based Heap Space Storage (ALCHSS), Destroy Activation Group-Based Heap Space (DESHS), Set Activation Group-Based Heap Space Storage Mark (SETHSSMK) and Materialize Activation Group-Based Heap Space Attributes (MATHSAT) instructions.

The heap space *creation template* identified by operand 2 must be 16-byte aligned in the space. Operand 2 is not modified by the instruction.

The following is the format of the heap space *creation template*:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Reserved (binary 0)	Char(8)	
8	8	Maximum single allocation	UBin(4)	
12	C	Minimum boundary alignment	UBin(4)	
16	10	Creation size	UBin(4)	
20	14	Extension size	UBin(4)	
24	18	Domain/Storage protection	Bin(2)	
		Hex 0000 = System should chose the domain		
		Hex 0001 = The heap space domain should be ″User″		
		Hex 8000 = The heap space domain should be ″System″		
26	1A	Heap space creation options	Char(6)	
26	1A		Allocation strategy	Bit 0
			0 = Normal allocation strategy	
			1 = Force process space creation on each allocate	
26	1A		Heap space mark	Bit 1
			0 = Allow heap space mark	
			1 = Prevent heap space mark	
26	1A		Block transfer	Bit 2

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = Transfer the minimum storage transfer size for this object	
			1 = Transfer the machine default storage transfer size for this object	
26	1A		Process access group member	Bit 3
			0 = Do not create the heap space in the PAG	
			1 = Create the heap space in the PAG	
26	1A		Allocation initialization	Bit 4
			0 = Do not initialize allocations	
			1 = Initialize allocations	
26	1A		Overwrite freed allocations	Bit 5
			0 = Do not overwrite freed allocations	
			1 = Overwrite freed allocations	
26	1A		Reserved (binary 0)	Bits 6-7
27	1B		Allocation value	Char(1)
28	1C		Freed value	Char(1)
29	1D		Reserved (binary 0)	Char(3)
32	20	Reserved (binary 0)	Char(64)	
96	60	—- End —-		

The **maximum single allocation** of any single allocation from the heap space is useful for controlling the use of the heap space and may also improve performance for some cases when the machine can optimize access based on this attribute. The minimum value that can be specified is 0 bytes, and the maximum value that can be specified is (16M - 1 page) bytes. To determine the current page size use the MATRMD instruction. If zero is specified, the default value of (16M - 1 page) bytes is used. Values outside the range indicated will cause a *template value invalid* (hex 3801) exception.

The **minimum boundary alignment** associated with any allocation from the heap space can be specified in the template as an advisory value. This value is expressed in terms of byte alignment. The machine will use the specified value to choose an actual alignment which is deemed closest to a machine-required alignment value. This allows changing machine requirements to be met without changing the advisory value. Storing valid pointers in heap space allocations will be supported for all advisory values, so the smallest effective alignment value is 16 byte alignment.

The **creation size** of the heap space can be specified in the template. If zero is specified, the system computes a default value. The minimum value that can be specified is 1 page (in bytes). The maximum value that can be specified is (16M - 1 page) bytes. To determine the current page size use the MATRMD instruction. The value specified is rounded up to a storage unit boundary. Values outside the range indicated cause a *template value invalid* (hex 3801) exception. This is an advisory value only. The machine may decide to override the value specified based on system resource constraints.

The **extension size** of the heap space can be specified in the template. If zero is specified, the system computes a default value. The minimum value that can be specified is 1 page (in bytes). The maximum value that can be specified is (16M - 1 page) bytes. To determine the current page size use the MATRMD instruction. The value specified is rounded up to a storage unit boundary. Values outside the range

indicated cause a *template value invalid* (hex 3801) exception. This is an advisory value only. The machine may decide to override the value specified based on system resource constraints.

The **domain/storage protection** field in the template allows the user of this instruction to override the domain for the heap space that would otherwise be chosen by the machine. The *domain/storage protection* attribute can be used to restrict access to the contents of the heap space by user state programs. It is possible to limit the access of the heap space by user state programs into 1 of two levels:

-
- No storage references (all storage references, modifying or non-modifying yield an *object domain or hardware storage protection violation* (hex 4401) exception). This is *system*.
- Full access (both modifying and non-modifying storage references are allowed). This is *user*.

Only a system state program can specify a heap space to be created with a domain of *system*. If a user state program attempts to specify the *domain/storage protection* as *system*, a *template value invalid* (hex 3801) exception will be signaled. Any value other than the ones listed will cause a *template value invalid* (hex 3801) exception to be signaled.

The *normal allocation strategy* as defined by the machine will be used unless the *force process space creation on each allocation* attribute is indicated. This option should only be used in unusual situations, such as when necessary for debug of application problems caused by references outside an allocation.

The **heap space mark** attribute can be used to prevent the use of the Set Activation Group-Based Heap Space Storage Mark (SETHSSMK) and Free Activation Group-Based Heap Space Storage from Mark(FREHSSMK) instructions on a heap space.

Block transfer on a heap space is used to increase the performance of a heap space based on prior knowledge of the program creating the heap space on how that heap space will be used. This attribute is used only when the heap space is not a member of a process access group (PAG).

A heap space can be created as a process access group (PAG) member of the process associated with the current thread, if specified by the **process access group member** field. It is possible for the PAG to overflow at which point any requested heap space creations or extensions will not reside in the PAG. Thus the specification to have the heap space as a member of the PAG is only an advisory which the machine may decide to override.

The **allocation initialization** field in the template allows the user of this instruction to specify that all storage allocations from the heap space being created will be *initialized* to the **allocation value** supplied in the template. If the user chooses *not to initialize* heap space storage allocations, the initial value of heap space storage allocations is unpredictable but will not expose data produced by a different user profile.

The **overwrite freed allocations** field in the template allows the user of this instruction to specify that all heap space storage allocations upon being freed *will be overwritten* with the **freed value** supplied in the template. If the user chooses *not to overwrite* heap space storage allocations when freed, the contents of the freed allocations will be unaltered.

A default heap space (heap identifier value of 0) is automatically available in each activation group, without issuing a Create Activation Group-Based Heap Space (CRTHS) instruction. The default heap space is created on the first allocation request of the default heap space. See Allocate Activation Group-Based Heap Space Storage (ALCHSS) for a description of the default heap space.

A heap space is scoped to an activation group, thus the maximum life of a heap space is the life of the activation group in which the heap space was created. A heap space can only be destroyed from within the activation group in which it was created.

Limitations (Subject to Change): The following are limits that apply to the functions performed by this instruction.

The amount of heap space storage that can be allocated for a single heap space is 4G-512K bytes. Due to fragmentation a heap space may grow to 4GB-512KB without having 4GB-512KB of outstanding heap space storage allocations.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Create Independent Index (CRTINX)

Op Code (Hex)	Operand 1	Operand 2
0446	Index	Index description template

Operand 1: System pointer.

Operand 2: Space pointer.

Bound program access

```
Built-in number for CRTINX is 34.
CRTINX (
        index                       : address of system pointer
        index_description_template  : address
)
```

Description: This instruction creates an independent index based on the index template specified by operand 2 and returns addressability to the index in a system pointer stored in the addressing object specified by operand 1.

The format of the *index description template* pointed to by operand 2 is as follows (must be aligned on a 16-byte multiple):

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4) +
4	4		Number of bytes available for materialization	Bin(4) +
8	8	Object identification	Char(32)	
8	8		Object type	Char(1) +
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Object creation options	Char(4)	
40	28		Existence attributes	Bit 0
			0 = Temporary	
			1 = Permanent	
40	28		Space attribute	Bit 1

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = Fixed-length	
			1 = Variable-length	
40	28		Initial context	Bit 2
			0 = Do not insert addressability in context	
			1 = Insert addressability in context	
40	28		Access group	Bit 3
			0 = Do not create as member of access group	
			1 = Create as member of access group	
40	28		Reserved (binary 0)	Bits 4-6
40	28		Initial owner specified	Bit 7
			0 = No	
			1 = Yes	
40	28		Reserved (binary 0)	Bits 8-12
40	28		Initialize space	Bit 13
			0 = Initialize	
			1 = Do not initialize	
40	28		Reserved (binary 0)	Bits 14-19
40	28		Always enforce hardware storage protection of this object	Bit 20
			0 = Enforce hardware storage protection of this object's encapsulated part only when hardware storage protection is being enforced for all storage.	
			1 = Enforce hardware storage protection of this object's encapsulated part at all times.	
40	28		Always enforce hardware storage protection of associated space	Bit 21
			0 = Enforce hardware storage protection of the associated space only when hardware storage protection is enforced for all storage.	
			1 = Enforce hardware storage protection of the associated space at all times.	
40	28		Reserved (binary 0)	Bits 22-31
44	2C	Recovery options	Char(4)	
44	2C		Reserved (binary 0)	Char(2)
46	2E		ASP number	Char(2)
48	30	Size of space	Bin(4)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
53	35		Space alignment	Bit 0
			0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If the *size of space* field is 0, this value must be specified.	
			1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space.	
			The value of this field is ignored when a value of 1 is given for the *machine chooses space alignment* field.	
53	35		Reserved (binary 0)	Bits 1-2
53	35		Machine chooses space alignment	Bit 3
			0 = The space alignment indicated by the *space alignment* field is performed.	
			1 = The machine will choose the space alignment most beneficial to performance, which may reduce maximum space capacity. When this value is specified, the *space alignment* field is ignored, but the alignment chosen will be a multiple of 512.	
53	35		Reserved (binary 0)	Bit 4
53	35		Main storage pool selection	Bit 5
			0 = Process default main storage pool is used for object.	
			1 = Machine default main storage pool is used for object.	
53	35		Reserved (binary 0)	Bit 6
53	35		Block transfer on implicit access state modification	Bit 7
			0 = Transfer the minimum storage transfer size for this object.	
			1 = Transfer the machine default storage transfer size for this object.	
53	35		Reserved (binary 0)	Bits 8-31
57	39	Reserved (binary 0)	Char(3)	
60	3C	Extension offset	Bin(4)	
64	40	Context	System pointer	
80	50	Access group	System pointer	
96	60	Index attributes	Char(1)	
96	60		Entry length attribute	Bit 0

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = Fixed-length entries	
			1 = Variable-length entries	
96	60		Immediate update	Bit 1
			0 = No immediate update	
			1 = Immediate update	
96	60		Key insertion	Bit 2
			0 = No insertion by key	
			1 = Insertion by key	
96	60		Entry format	Bit 3
			0 = Scalar data only	
			1 = Both pointers and scalar data	
96	60		Optimized processing mode	Bit 4
			0 = Optimize for random references	
			1 = Optimize for sequential references	
96	60		Maximum entry length (obsolete)	Bit 5
96	60		Index coherency tracking	Bit 6
			0 = Do not track index coherency	
			1 = Track index coherency	
96	60		Longer template	Bit 7
			0 = The template is the original size	
			1 = The template is longer	
97	61	Argument length	Bin(2)	
99	63	Key length	Bin(2)	
101	65	—- End —-		

Note: This instruction ignores the values associated with the fields annotated with a plus sign (+).

The template identified by operand 2 must be 16-byte aligned.

There are two ways the operand 2 template can be extended. When the **longer template** field is set to binary 1, the fields starting at offset 101 in the longer template are defined. Also, if the *extension offset* is non-zero, a template extension is located by the *extension offset* field.

If the *longer template* field is set to binary 1, then the longer template is defined starting at offset 101 of the operand 2 template. The longer template is defined as follows:

Offset Dec	Hex	Field Name	Data Type and Length
101	65	Reserved (binary 0)	Char(12)
113	71	Template version	Char(1)

Offset Dec	Hex	Field Name	Data Type and Length
114	72	Index format	Char(1)
		0 = Maximum object size of 4 Gigabytes.	
		1 = Maximum object size of 1 Terabyte.	
115	73	Reserved (binary 0)	Char(61)
176	B0	—- End —-	

If the **extension offset** is non-zero, a template extension is defined at this offset from the beginning of the operand 2 template. The template extension must be 16-byte aligned in the space. The following is the format of the template extension:

Offset Dec	Hex	Field Name	Data Type and Length
0	0	User profile	System pointer
16	10	Reserved (binary 0)	Char(4)
20	14	Domain assigned to the object	Char(2)
22	16	Reserved (binary 0)	Char(42)
64	40	—- End —-	

If the created object is permanent, it is owned by the user profile governing thread execution. The owning user profile is implicitly assigned all private authority states for the object. The storage occupied by the created object is charged to this owning user profile. If the created object is temporary, there is no owning user profile, and all authority states are assigned as public. Storage occupied by the created object is charged to the process associated with the creating thread.

The **object identification** specifies the symbolic name that identifies the space within the machine. An **object type** of hex 0E is implicitly supplied by the machine. The *object identification* is used to identify the object on materialize instructions as well as to locate the object in a context that addresses the object.

The **existence attribute** specifies that the index is to be created as a *permanent* or a *temporary* object. A temporary index, if not explicitly destroyed by the user, is implicitly destroyed by the machine when machine processing is terminated.

Permanent index objects cannot be created by user state programs when the system security level is 40 or above.

A space may be associated with the created object. The space may be *fixed* or *variable* in size, as specified by the **space attribute** field. The initial allocation is as specified in the **size of space** field. The machine allocates a space of at least the size specified. The actual size allocated is dependent on an algorithm defined by a specific implementation.

If the **initial context** creation attribute field indicates that *addressability is to be inserted* in a context, the **context** field must be a system pointer that identifies a context where addressability to the newly created object is to be placed. If the *initial context* indicates that *addressability is not to be placed in a context*, the *context* field is ignored.

If the **access group** creation attribute field indicates that the object *is to be created in an access group*, the **access group** field must be a system pointer that identifies an access group in which the object is to be created. The *existence attribute* must be *temporary* for an object to be in an access group. If the object *is not to be created in an access group*, the *access group* field is ignored.

The **initial owner specified** creation option controls whether or not the initial owner of the independent index is to be the user profile specified in the template. When *yes* is specified, initial ownership is assigned to the user profile specified in the **user profile** field of the template extension. When *no* is specified, initial ownership is assigned to the user profile governing thread execution. The initial owner user profile is implicitly assigned all authority states for the object. The storage occupied by the object is charged to the initial owner.

The **initialize space** creation option controls whether or not the space is to be initialized. When *initialize* is specified, each byte of the space is initialized to a value specified by the **initial value of space** field. Additionally, when the space is extended in size, this byte value is also used to initialize the new allocation. When *do not initialize* is specified, the *initial value of space* field is ignored and the initial value of the bytes of the space are unpredictable.

When *do not initialize* is specified for a space, internal machine algorithms do ensure that any storage resources last used for allocations to another object which are reused to satisfy allocations for the space are reset to a machine default value to avoid possible access of data which may have been stored in the other object. To the contrary, reusage of storage areas previously used by the space object are not reset, thereby exposing subsequent reallocations of those storage areas within the space to access of the data which was previously stored within them.

The **always enforce hardware storage protection of this object** field is used to specify whether the hardware storage protection defined by the machine for this object type should be enforced at all times, or only when hardware storage protection is enforced for all storage. This option applies to all the encapsulated storage associated with this object, but not to the associated space, if any.

The **always enforce hardware storage protection of associated space** field is used to specify whether the hardware storage protection defined by the machine for this object type's associated space should be enforced at all times, or only when hardware storage protection is enforced for all storage.

The **ASP number** field specifies the ASP number of the ASP on which the object is to be allocated. A value of 0 indicates an *ASP number* is not specified and results in the default of allocating the object in the system ASP. Allocation on the system ASP can only be done implicitly by not specifying an *ASP number*. The only nonzero values allowed are 2 through 255 which provide for explicit allocation of objects on a user ASP or an independent ASP. The ASP number must specify an existing ASP. An *ASP number* of 1 or greater than 255 results in a *template value invalid* (hex 3801) exception being signalled. The given *ASP number* must be currently configured on the system otherwise an *auxiliary storage pool number invalid* (hex 1C09) exception is signalled. If the ASP number identifies an independent ASP, there must be an existing active logical unit description for the independent ASP otherwise an *independent asp varied off* (hex 1C11) exception is signalled. A temporary object cannot be created in an ASP other than the system ASP. If this is attempted, a *template value invalid* (hex 3801) exception is signalled. If the ASP number identifies an independent ASP, or *initial context* indicates that addressability is to be inserted into a context that resides in an independent ASP, then both must indicate the same independent ASP or belong to the same ASP group or else a *template value invalid* (hex 3801) exception is signalled. The *ASP number* of an object can be materialized, but cannot be modified.

The **performance class** field provides information allowing the machine to more effectively manage the object considering the overall performance objectives of operations involving the index.

If the **entry length attribute** field specifies *fixed-length entries*, the entry length of every index entry is established at creation by the value in the **argument length** field of the index description template. If the *entry length* attribute field specifies *variable-length entries*, then entries will be variable-length (the length of each entry is supplied when the entry is inserted), and the *argument length* field is ignored.

If the **immediate update** field specifies that an *immediate update* should occur, then every update to the index will be written to auxiliary storage after every insert or remove operation. This option is ignored if

the *existence attributes* field is set to binary 0 (temporary). In this case, the value of the *immediate update* field is assumed to be binary 0 and updates will not be written to auxiliary storage after every insert or remove operation.

If the **key insertion** field specifies *insertion by key*, then the **key length** field must be specified. This allows the specification of a portion of the argument (the key), which may be manipulated in either of the following ways in the Insert Index Entry (INSINXEN) instruction:

•
- The insert will not take place if the key portion of the argument is already in the index.
- The insert will cause the nonkey portion of the argument to be replaced if the key is already in the index.

The **entry format** field designates the index entries as containing *both pointers and scalar data* or *scalar data only*. The *both pointers and scalar data* field can be used only for indexes with fixed-length entries. If the index is created to contain both pointers and data, then

•
- Entries to be inserted must be 16-byte aligned.
- Each entry retrieved by the Find Independent Index Entry (FNDINXEN) instruction or the Remove Independent Index Entry (RMVINXEN) instruction is 16-byte aligned.
- Pointers are allowed in both the key and nonkey portions of an index entry.
- Pointers need not be at the same location in every index entry.
- Pointers inserted into the index remain unchanged. No resolution is performed before insertion.

If the index is created to contain *scalar data only*, then:

•
- Entries to be inserted need not be aligned.
- Entries returned by the Find Independent Index Entry (FNDINXEN) instruction or the Remove Independent Index Entry (RMVINXEN) instruction are not aligned.
- Any pointers inserted into the index will be invalidated.

The **optimized processing mode** index attribute field is used to designate whether the index should be created and maintained in a manner that optimizes performance for either *random* or *sequential* operations.

The **maximum entry length** field is ignored. All indexes are created with a maximum entry length of 2,000 bytes. Note that indexes created before Version 3 Release 6 could have been created with a *maximum entry length* of 120 bytes or 2,000 bytes. The MATINXAT instruction can be used to materialize this attribute.

The **key length** field specifies the length of the key for the entries that are inserted into the index. The **argument length** specifies the length of the entries when *fixed length* entries are used.

The *key length* must have a value less than or equal to the *argument length* whether specified during creation (for fixed-length entries) or during insertion (for variable length). The *key length* is not used if the *key insertion field* specifies *no insertion by key*.

The field **template version** identifies the version of the longer template. It must be set to hex 00.

The **index format** field determines the format of the index. This attribute cannot be modified after the index has been created. If an index is created with a format of hex 01 (maximum object size of 1 terabyte), the index cannot be saved to a target release earlier than Version 5 Release 2. If the longer template is not defined (i.e. the field *longer template* is set to binary 0), the *index format* field defaults to a value of hex 00 (maximum object size of 4 gigabytes).

If the **index coherency tracking** attribute field specifies *track index coherency* then additional checking is done when the index is referenced for the first time after an IPL to determine if the index was coherent at system termination. If the index is found to not be coherent then the index is marked as damaged. The index is not coherent while the internal structure is being modified, for example, during an insert or a remove operation.

If the system is not able to save its main storage at system termination then it can not be determined whether or not the index is coherent and the index is marked as damaged the next time it is referenced.

If the *index coherency tracking* attribute field specifies *do not track index coherency* then no additional checking is done when the index is referenced for the first time after an IPL.

The **extension offset** specifies the byte offset from the beginning of the operand 2 template to the beginning of the template extension. An offset value of zero specifies that the template extension is not provided. A negative offset value is invalid. A non-zero offset must be a multiple of 16 (to cause 16-byte alignment of the extension). Except for these restrictions, the offset value is not verified for correctness relative to the location of other portions of the create template.

The **domain assigned to the object** field in the template extension allows the user of this instruction to override the domain for this object that would otherwise be chosen by the machine. Valid values for this field are:

Domain field	Domain assigned to the object
Hex 0000	The domain will be chosen by the machine.
Hex 0001	The domain will be 'User'.

Any value specified for the *domain assigned* field other than those listed above will result in a *template value invalid* (hex 3801) exception being signalled.

Limitations (Subject to Change): The following are limits that apply to the functions performed by this instruction. These limits may change on different implementations of the machine.

The size of the object specific portion of the object is limited to a maximum of 4 gigabytes or 1 terabyte, depending on the value of the *index format* field. This size is dependent upon the amount of storage needed for the number and size of index entries and excludes the size of the associated space, if any.

The size of the associated space for this object is limited to a maximum of 16MB-32 bytes if the machine does not choose the space alignment and 0 is specified for the *space alignment* field. The size of the associated space for this object is limited to a maximum of 16MB-512 bytes if the machine does not choose the space alignment and 1 is specified for the *space alignment* field. The maximum size of an associated space for this object if the machine chooses the space alignment is returned by option Hex 0003 of MATMDATA.

Warning: Temporary Level 3 Header

Authorization Required

-
- Insert
  -
  - Context identified by operand 2
  - User profile of object owner
- Execute
  -
  - Contexts referenced for address resolution

Lock Enforcement

•
- Modify
  –
  – Access group identified by operand 2
  – User profile of object owner
  – Context identified by operand 2
- Materialize
  –
  – Contexts referenced for address resolution

Exceptions

02 Access Group

0201 Object Ineligible for Access Group

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

0E Context Operation

0E01 Duplicate Object Identification

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C09 Auxiliary Storage Pool Number Invalid

1C0E IASP Resources Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

4403 Cannot Change Contents of Protected Context

Create Pointer-Based Mutex (CRTMTX)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03C3	Mutex	Creation template	Result

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Signed binary(4) variable scalar.

Bound program access

```
Built-in number for CRTMTX is 161.
CRTMTX (
        mutex               : address
        creation_template   : address
) :   signed binary(4)  /* result */
```

Note: The term ″mutex″ in this instruction refers to a ″pointer-based mutex″.

Description: A mutex is created and associated with the storage location of the *mutex* whose address is passed in operand 1. The *mutex* is initialized into the unlocked state so it may be used for mutual exclusion between threads attempting to lock the *mutex*. An *optional name string* can be associated with the *mutex*. A mutex must be created before it can be used for synchronization with the Lock Pointer-Based Mutex (LOCKMTX) and Unlock Pointer-Based Mutex (UNLKMTX) instructions, and with other mutex instructions such as Destroy Pointer-Based Mutex (DESMTX) and Materialize Mutex (MATMTX).

Result is used to indicate the success or failure of the CRTMTX instruction. Following a successful CRTMTX, the *mutex* is used by passing its address as a parameter to other mutex instructions.

Mutexes are temporary entities that do not persist beyond the current IPL in which they are created. Following a subsequent IPL, mutexes must be re-created before they can be used. Similarly, mutexes created in an independent ASP do not persist beyond the current vary on of the independent ASP in which they are created. Following a subsequent vary on, mutexes must be re-created before they can be used. Mutexes can be explicitly destroyed prior to a subsequent IPL or independent ASP vary off/vary on cycle by using the DESMTX instruction.

It is important to destroy mutexes when they are no longer needed. When a mutex is created, system resources are allocated for the mutex. These resources remain allocated until the mutex is destroyed (see DESMTX instruction) or the system is IPLed, thereby leaving fewer mutex resources available in the system for other threads to use. In addition, performance degradation can occur as unused mutexes accumulate on the system and are not destroyed.

The *mutex* must be aligned on a 16-byte boundary. The format is as follows:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Mutex control area	Char(16)
16	10	Optional name string	Char(16)
32	20	—- End —-	

The **mutex control area** contains a synchronization pointer. This field is initialized by the CRTMTX instruction, and is used only by the machine. The caller of CRTMTX should simply define the *mutex control area* as a 16-byte character field.

The *mutex* must be at least 16 bytes in length. If the mutex *creation template* associates a name with the *mutex*, then the *optional name string* must reside in the storage immediately following the 16-byte *mutex control area* and be initialized prior to the CRTMTX call. The **optional name string** is a character string which can be a maximum of 16 bytes long. The name string must either be null-terminated (up to 15 bytes of data followed by a null byte), or it must be 16 bytes long, padded with blanks. If a null byte is found within the first 16 bytes of the name string, the name string is considered to be null-terminated. No validity checking is performed to ensure mutex names are unique in the system, and it is possible to have multiple mutexes with the same name; however, it is recommended that each mutex be given a unique name. This name cannot be used on any mutex instructions to identify the mutex; rather, it is used mainly to be able to identify the mutex more easily during problem determination. The mutex name, if specified, is returned from the MATMTX instruction.

The *creation template* specified by operand 2 is used to initialize the *mutex*. The format is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Reserved (binary 0)	Char(1)
1	1	Name option	Char(1)
		Hex 00 = No name string is associated with this mutex (this is the default option when the template is not supplied).	
		Hex 01 = Mutex has a name string associated with it. Name string immediately follows the mutex in the template.	
2	2	Keep valid option	Char(1)
		Hex 00 = Mutex will be destroyed when its owning thread is terminated (this is the default option when the template is not supplied).	
		Hex 01 = Mutex will remain valid when its owning thread is terminated. The mutex will be marked as being in a pending state.	
3	3	Recursive option	Char(1)
		Hex 00 = Recursive attempts to lock this mutex will not be permitted (this is the default option when the template is not supplied).	
		Hex 01 = Recursive attempts to lock this mutex will be permitted by the same thread that has already locked the mutex.	
4	4	Reserved (binary 0)	Char(28)
32	20	—- End —-	

The **name option** field specifies whether the *mutex* is to be named. The *name option* field must be set to hex 01 in order to associate a name with the *mutex*. Setting the *name option* to hex 00 will create a mutex without a name. All other values for *name option* are reserved.

Keep valid option field specifies whether the *mutex* is to remain valid when a thread is terminated while holding the lock on the *mutex*. The *keep valid option* field must be set to hex 01 to indicate that the *mutex* should remain valid after thread termination. The *mutex* is considered to be in a pending state when it is kept valid after thread termination. The next thread to lock a pending mutex will revalidate the mutex, but will receive an EUNKNOWN error number to indicate that the resource protected by this mutex may require special handling. Appropriate action is left up to the MI user's discretion. Specifying hex 00 for the *keep valid option* field will result in the *mutex* being destroyed during thread termination. Threads waiting for the *mutex* that was locked by the terminating thread will be returned the EOWNERTERM error number.

The **recursive option** field specifies whether recursive locking of the *mutex* will be allowed. See LOCKMTX for additional information on the recursive behavior of a pointer-based mutex.

If operand 2 in a bound program is a null pointer value, the *mutex* will be created with the default initialization options. Operand 2 in non-bound programs must be a pointer to a *creation template*. The *pointer does not exist* (hex 2401) exception will be signaled if a null pointer value is used for operand 2 in a non-bound program.

If the *mutex* is created by this instruction, then *result* is set to 0. If an error occurs, then the *result* is set to an error number. The EINVAL error number will be returned if an invalid parameter is specified. The EPERM error number is returned when the address passed in operand 1 is in teraspace and the issuing thread does not have teraspace write permissions to that address. The ENOMEM error number is returned when no more mutexes can be created due to lack of system resources.

An attempt to create a new mutex using the same space as an existing mutex will cause the existing mutex to be destroyed. An attempt to create a new mutex using the same space as a copy of an existing mutex will not cause the existing mutex to be destroyed, instead, a new mutex will be created in the space where the copy resides.

Programming Considerations: Although copies of an existing mutex can be made by other instructions that copy memory and preserve pointers, or can effectively be made by teraspace memory mapping techniques, making and using mutex copies is not recommended. A copy of a mutex has certain restrictions, for example, destroying a mutex by means of a copy of the mutex is not permitted. Therefore, it is necessary that the mutex originally created by CRTMTX remain intact until it is no longer needed. Additional restrictions may apply to mutex copies in other releases.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Error conditions

The *result* will be set to one of the following:

EINVAL 3021 - The value specified for the argument is not correct.

ENOMEM 3460 - Storage allocation request failed.

EPERM 3027 - Operation not permitted.

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

36 Space Management

- 3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Create Space (CRTS)

Op Code (Hex)	Operand 1	Operand 2
0072	Pointer for space addressability	Creation template

Operand 1: System pointer.

Operand 2: Space pointer.

Bound program access

```
Built-in number for CRTS is 25.
CRTS (
      pointer_for_space_addressability   : address of system pointer
      creation_template                  : address
)
```

Description: A space object is created with the attributes that are specified in the space *creation template* specified by operand 2, and addressability to the created space is placed in a system pointer that is returned in the addressing object specified by operand 1.

Space objects, unlike other types of system objects, are used to contain a space and serve no other purposes.

The template identified by operand 2 must be 16-byte aligned in the space. The following is the format of the space *creation template*:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size specification	Char(8) +	
0	0		Size of template	Bin(4) +
4	4		Number of bytes available for materialization	Bin(4) +
8	8	Object identification	Char(32)	
8	8		Object type	Char(1) +
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Object creation options	Char(4)	
40	28		Existence attribute	Bit 0
			0 = Temporary	
			1 = Permanent	
40	28		Space attribute	Bit 1
			0 = Fixed-length	
			1 = Variable-length	
40	28		Initial context	Bit 2

Offset Dec	Offset Hex	Field Name	Data Type and Length		
			0 =	Addressability is not inserted into context	
			1 =	Addressability is inserted into context	
40	28		Access group		Bit 3
			0 =	Do not create as member of access group	
			1 =	Create as member of access group	
40	28		Reserved (binary 0)		Bits 4-5
40	28		Public authority specified		Bit 6
			0 =	No	
			1 =	Yes	
40	28		Initial owner specified		Bit 7
			0 =	No	
			1 =	Yes	
40	28		Reserved (binary 0)		Bits 8-11
40	28		Set public authority in operand 1		Bit 12
			0 =	No	
			1 =	Yes	
40	28		Initialize space		Bit 13
			0 =	Initialize	
			1 =	Do not initialize	
40	28		Automatically extend space		Bit 14
			0 =	No	
			1 =	Yes	
40	28		Hardware storage protection level		Bits 15-16
			00 =	Reference and modify allowed for user state programs	
			01 =	Only reference allowed for user state programs	
			10 =	Invalid (yields *template value invalid* (hex 3801) exception) The MODS instruction can be used to change the hardware storage protection level to 10.	
			11 =	No reference or modify allowed for user state programs	
40	28		Process temporary space accounting		Bit 17

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = The temporary space will be tracked to the creating process	
			1 = The temporary space will not be tracked to the creating process	
40	28		Reserved (binary 0)	Bits 18-20
40	28		Always enforce hardware storage protection of space	Bit 21
			0 = Enforce hardware storage protection of this space only when hardware storage protection is being enforced for all storage.	
			1 = Always enforce hardware storage protection of this space.	
40	28		Reserved (binary 0)	Bits 22-31
44	2C	Recovery options	Char(4)	
44	2C		Reserved (binary 0)	Char(2)
46	2E		ASP number	Char(2)
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
53	35		Space alignment	Bit 0
			0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If the *size of space* field is zero, this value must be specified.	
			1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space.	
			The value of this field is ignored when a value of 1 is given for the *machine chooses space alignment* field.	
53	35		Clear the space into main memory during creation	Bit 1
			0 = Only a minimum amount (up to 4K) of the space will be in main storage upon completion of the instruction.	
			1 = Most of the space, with some limits enforced by the machine, will be in main storage upon completion of the instruction.	
53	35		Spread the space object among storage devices	Bit 2

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = The space should be created on one storage device, if possible.	
			1 = The space should be created spread among available storage devices, if possible. All extensions to the space object will also be spread unless this attribute is changed using the Modify Space Attributes (MODS) instruction.	
53	35		Machine chooses space alignment	Bit 3
			0 = The space alignment indicated by the *space alignment* field is performed.	
			1 = The machine will choose the space alignment most beneficial to performance, which may reduce maximum space capacity. When this value is specified, the *space alignment* field is ignored, but the alignment chosen will be a multiple of 512. The maximum capacity for a space object for which the machine has chosen the alignment is returned by option Hex 0003 of MATMDATA. The maximum space capacity for a particular space object is returned by MATS.	
53	35		Reserved (binary 0)	Bit 4
53	35		Main storage pool selection	Bit 5
			0 = Process default main storage pool is used for object.	
			1 = Machine default main storage pool is used for object.	
53	35		Transient storage pool selection	Bit 6
			0 = Default main storage pool (process default or machine default as specified for main storage pool selection) is used for object.	
			1 = Transient storage pool is used for object.	
53	35		Obsolete	Bit 7
			This field is no longer used and will be ignored.	
53	35		Unit number	Bits 8-15 +
53	35		Reserved (binary 0)	Bits 16-23
56	38		Expanded transfer size advisory	Char(1)
57	39	Reserved (binary 0)	Char(1)	
58	3A	Public authority	Char(2)	
60	3C	Extension offset	Bin(4)	
64	40	Context	System pointer	
80	50	Access group	System pointer	
96	60	—- End —-		

Note: The instruction ignores the values associated with template fields annotated with a plus sign (+).

A template extension must be specified for the *initial owner specified* creation option. Also, the template extension must be specified (*extension offset* must be nonzero) to specify any of the other template extension fields (those other than the initial owner user profile) as input to the instruction.

The template extension is located by the *extension offset* field. The template extension must be 16-byte aligned in the space. The following is the format of the template extension:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	User profile	System pointer
16	10	Largest size needed for space	Bin(4)
20	14	Domain assigned to the object	Char(2)
		Hex 0000 = The domain will be chosen by the machine.	
		Hex 0001 = The domain will be 'User'.	
		Hex 8000 = The domain will be 'System'.	
22	16	Reserved (binary 0)	Char(42)
64	40	—- End —-	

If the created object is permanent, it is owned by the user profile governing thread execution. The owning user profile is implicitly assigned all private authority states for the object. The storage occupied by the created object is charged to this owning user profile. If the created object is temporary, there is no owning user profile, and all authority states are assigned as public. Storage occupied by the created space is charged to the process.

Permanent space objects cannot be created by user state programs when the system security level is 40 or above.

The **object identification** specifies the symbolic name that identifies the space within the machine. An **object type** of hex 19 is implicitly supplied by the machine. The *object identification* is used to identify the object on materialize instructions as well as to locate the object in a context that addresses the object. The *object subtype* must be hex EF.

The **existence attribute** specifies whether the space is to be created as temporary or permanent. A *temporary* space, if not explicitly destroyed by the user, is implicitly destroyed by the machine when machine processing is terminated. A *permanent* space exists in the machine until it is explicitly destroyed by the user.

The **space attribute** specifies whether the size of the space can vary. The space may have a *fixed* size or a *variable* size. The initial allocation is as specified in the **size of space** field. The machine allocates a space of at least the size specified. The actual size allocated depends on an algorithm defined by a specific implementation. A fixed size space of zero length causes a *template value invalid* (hex 3801) exception to be signaled.

If the **initial context** creation attribute field indicates that *addressability is inserted into context*, the **context** field must contain a system pointer that identifies a context where addressability to the newly created space is to be placed. If *addressability is not inserted into a context*, the *context* field is ignored.

If the **access group** creation attribute field indicates that the space is to be created in an access group, the *access group* field must be a system pointer that identifies the access group in which the space is to be created. If the space is being *created as a member of an access group*, the *existence attribute* field must be *temporary* (bit 0 equals 0). If the space *is not to be created into an access group*, the *access group* field is ignored.

The **expanded transfer size advisory** specifies the desired number of pages to be transferred between main store and auxiliary storage for implicit access state changes. This value is only an advisory; the machine may use a value of its choice for performing access state changes under some circumstances. For example, the machine may limit the transfer size to a smaller value than is specified. A value of zero is an explicit indication that the machine should use the machine default storage transfer size for this object.

The **public authority specified** creation option controls whether or not the space is to be created with the public authority specified in the template. When *yes* is specified, the space is created with the public authority specified in the **public authority** field of the template. When *no* is specified, the *public authority* field is ignored and the space is created with default public authority. The default public authority depends on the value of the *existence attribute*: An *existence attribute* value of *temporary* results in a default public authority of all authority; an *existence attribute* value of *permanent* results in a default public authority of no authority.

The **initial owner specified** creation option controls whether or not the initial owner of the space is to be the user profile specified in the template. When *yes* is specified, initial ownership is assigned to the user profile specified in the **user profile** field of the template extension. When *no* is specified, initial ownership is assigned to the thread user profile and the *user profile* field in the template extension is ignored. The initial owner user profile is implicitly assigned all authority states for the object. The storage occupied by the created space is charged to the initial owner. If *yes* is specified for this creation option when the *existence attribute* specifies *temporary*, a *template value invalid* (hex 3801) exception will be signaled.

The **set public authority in operand 1** creation option controls, when the *public authority specified* creation option has also been specified as *yes*, whether or not the public authority attribute for the space is to be set into the system pointer returned in operand 1. When *yes* is specified, the specified *public authority* is set into operand 1. When *no* is specified, public authority is not set into operand 1. When the *public authority specified* creation option is set to *no*, this option can not be specified as *yes* (or else a *template value invalid* (hex 3801) exception will be signalled) and the authority set into operand 1 is the default of no authority for a *permanent* or all authority for a *temporary* object (as specified by the *existence attribute*).

The **initialize space** creation option controls whether or not the space is to be initialized. When *initialize* is specified, each byte of the space is initialized to a value specified by the **initial value of space** field. Additionally, when the space is extended in size, this byte value is also used to initialize the new allocation. When *do not initialize* is specified, the *initial value of space* field is ignored and the initial value of the bytes of the space are unpredictable.

When *do not initialize* is specified for a space, internal machine algorithms do ensure that any storage resources last used for allocations to another object which are reused to satisfy allocations for the space are reset to a machine default value to avoid possible access of data which may have been stored in the other object. To the contrary, reusage of storage areas previously used by the space object are not reset, thereby exposing subsequent reallocations of those storage areas within the space to access of the data which was previously stored within them.

The **automatically extend space** creation option controls whether the space is to be extended automatically by the machine or a *space addressing violation* (hex 0601) exception is to be signaled when a reference is made to an area beyond the allocated portion of the space. When *yes* is specified, the space will automatically be extended by an amount determined through internal machine algorithms. When *no* is specified, the exception will result. Note that an attempt to reference an area beyond the maximum size that a space can be allocated, will always result in the signaling of the *space addressing violation* (hex

0601) exception independently of the setting of this attribute. The *automatically extend space* creation option can only be specified when the *space attribute* has been specified as *variable length*. Invalid specification of the *automatically extend space* option results in the signaling of the *template value invalid* (hex 3801) exception.

Usage of the *automatically extend space function* is limited. Predictable results will occur only when you ensure that the automatic extension of a space will not happen in conjunction with modification of the space size by another thread. That is, you must ensure that when a thread is using the space in a manner that could cause it to be automatically extended, it is the sole thread which can cause the space size to be modified. Note that in addition to implicit modification through automatic extension, the space size can be explicitly modified through use of the Modify Space Attributes (MODS) instruction.

The **hardware storage protection level** can be used to restrict access to the contents of the space by user state programs. It is possible to limit the access of the space by user state programs into 1 of three levels:

-
- *Reference only* (non-modifying storage references are allowed, modifying storing storage references yield an *object domain or hardware storage protection violation* (hex 4401) exception).
- *No storage references* (all storage references, modifying or non-modifying yield an *object domain or hardware storage protection violation* (hex 4401) exception).
- *Full access* (both modifying and non-modifying storage references are allowed).

Process temporary space accounting can be used to detect when temporary space objects, created within a process, still exist at process termination time. Temporary spaces that are created with the *process temporary space accounting* field set to 0 will be "tracked" to the process which created them. Temporary spaces that are created with the *process temporary space accounting* field set to 1 will not be "tracked" to the creating process.

At process termination time, any tracked spaces that exist may cause the machine to attempt to destroy the existing tracked spaces. If this is done, the destroy attempts would be performed as if an MI program issued a Destroy Space (DESS) instruction for each of the existing spaces.

The purpose of *process temporary space accounting* is to identify objects which may be "lost" in the system (until the next IPL). It should not intentionally be used (by MI) as a method of cleaning up temporary space objects at process termination time. The machine does not guarantee that all spaces (that should be tracked) will indeed be tracked. Also, if the machine is attempting to destroy tracked spaces at process termination time, any failures in the deletion attempts (such as if a space is locked to another process) will be ignored (i.e. the space will not be destroyed) and no indication of this is presented to the MI user.

Process temporary space accounting only applies to temporary space objects. A value of 1 for the *process temporary space accounting* field when creating a permanent object will result in a *template value invalid* (hex 3801) exception. This is in spite of the fact that a value of 1 for this field would result in the same actions as when creating a permanent object (i.e. the object would not be tracked to the process). The exception is presented because this field is undefined for permanent objects.

The **always enforce hardware storage protection of space** field is used to specify whether the hardware storage protection given in the *hardware storage protection level* field should be enforced at all times for this space, or only when hardware storage protection is being enforced for all storage.

The **ASP number** field specifies the ASP number of the ASP on which the unit is to be allocated. A value of 0 indicates an *ASP number* is not specified and results in the default of allocating the object in the system ASP. Allocation on the system ASP can only be done implicitly by not specifying an ASP number. The only non-zero values allowed are 2 through 255 which specify the user ASP on which the space object will be allocated. Independent ASPs have numbers 33 through 255. The *ASP number* must specify an existing ASP. An *ASP number* of 1 or greater than 255 results in a *template value invalid* (hex 3801) exception being signalled. The given *ASP number* must be currently configured on the system otherwise

an *auxiliary storage pool number invalid* (hex 1C09) exception is signalled. If the ASP number identifies an independent ASP, there must be an existing active logical unit description for the independent ASP otherwise an *independent asp varied off* (hex 1C11) exception is signalled. A temporary object cannot be created into a user ASP. If this is attempted, a *template value invalid* (hex 3801) exception is signalled. If the ASP number identifies an independent ASP, or *initial context* indicates that addressability is to be inserted into a context that resides in an independent ASP, then both must indicate the same independent ASP or belong to the same ASP group or else a *template value invalid* (hex 3801) exception is signalled. The *ASP number* attribute of an object can be materialized, but cannot be modified.

The **size of space** field specifies the amount of user-addressable storage being requested for this create.

The **performance class** fields provide information allowing the machine to more effectively manage the space object considering the overall performance objectives of operations involving the space.

The **extension offset** specifies the byte offset from the beginning of the operand 2 template to the beginning of the template extension. An offset value of zero specifies that the template extension is not provided. A negative offset value is invalid. A non-zero offset must be a multiple of 16 (to cause 16 byte alignment of the extension). Except for these restrictions, the offset value is not verified for correctness relative to the location of other portions of the create template.

The **largest size needed for space** field of the template extension specifies, when nonzero, a value in bytes that indicates the largest size that will be needed for the space. This field is different from the *size of space* field which indicates the size for the initial allocation of the space. This field can be used to communicate to the machine what the largest size needed for the space will be. Specification of a value larger than the maximum size space allowed for the space alignment chosen is invalid and results in signaling of the *template value invalid* (hex 3801) exception. Specification of a nonzero value that is less than the *size of space* field also results in the signaling of the *template value invalid* (hex 3801) exception. For more information on the maximum allowed, see "Limitations (Subject to Change)" (page 292).

Specifying the *largest size needed for space* value allows the machine, under certain circumstances, to select usage of an internal storage allocation unit which best utilizes the internal addressing resources within the machine. Note that the internal storage allocation unit selected can alter the maximum modification size of the associated space for the object. However, the machine will always use an internal storage allocation unit that will allow for extension of the space to at least the value specified in the *largest size needed for space* field. The maximum size to which the space can be modified is dependent upon specific implementations of the machine and can vary with different machine implementations. For more information on the effect of this option, see "Limitations (Subject to Change)" (page 292).

The **domain assigned** field in the template extension allows the user of this instruction to override the domain for this object that would otherwise be chosen by the machine.

Any value specified for the *domain assigned* field other than those listed will result in a *template value invalid* (hex 3801) exception being signalled.

Limitations (Subject to Change): The following are limits that apply to the functions performed by this instruction. These limits may change on different implementations of the machine.

The maximum size of any space object for which the machine chose the alignment is returned by option Hex 0003 of MATMDATA. The maximum size of a particular space object is returned by the MATS instruction.

Warning: Temporary Level 3 Header

Authorization Required

•
- Insert
  –
  – User profile of object owner
  – Context identified in operand 2
- Execute
  –
  – Context referenced for address resolution
- Object control
  –
  – Operand 1 if being replaced

Lock Enforcement

•
- Materialize
  –
  – Contexts referenced for address resolution
- Modify
  –
  – Context identified in operand 2
  – User profile of object owner
  – Access group identified in operand 2

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

0E Context Operation

0E01 Duplicate Object Identification

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C09 Auxiliary Storage Pool Number Invalid

1C0E IASP Resources Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

4403 Cannot Change Contents of Protected Context

Deactivate Program (DEACTPG)

Op Code (Hex)	Operand 1
0225	Program

Operand 1: System pointer or null.

Bound program access
Built-in number for DEACTPG is 33. DEACTPG (program : address of system pointer OR null operand)

Description: This instruction, provided that certain conditions are met, deactivates a non-bound program or a bound program activated for non-bound program compatibility. Subsequent invocations of the *program* within the same activation group will cause a new activation to be created.

Operand 1 specifies a program activation entry which is to be deactivated, if permitted. The activation entry is inferred by one of two means:

1. *operand 1 is null* — the target activation entry is that associated with the current invocation
2. *operand 1 is not null* — the target activation entry associated with the *program* system pointer is selected from one of the two default activation groups

The target activation entry is deactivated if permitted. An *activation in use by invocation* (hex 2C05) exception is signaled if the deactivation is not permitted. If the target activation entry does not exist, then no operation is performed. If the *program* specified is a bound program and not activated like a non-bound program, an *invalid operation for program* (hex 2C15) exception is signaled.

In general, only those activations with a zero invocation count can be deactivated. The following two exceptions apply:

1. A program can deactivate itself if it is the only invocation of that program in the process (its invocation count must be 1.)
2. An invocation exit program can deactivate the program on whose behalf it is running provided that the invocation count of that program is no more than 1.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

–
– Contexts referenced for address resolution

Lock Enforcement

•
• Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2C Program Execution

2C05 Activation in Use by Invocation

2C15 Invalid Operation for Program

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Decompress Data (DCPDATA)

Op Code (Hex)	Operand 1
1051	Decompress data template

Operand 1: Space pointer.

Bound program access
Built-in number for DCPDATA is 108. DCPDATA (decompress_data_template : address)

Description: The instruction decompresses user data. Operand 1 identifies a template which identifies the data to be decompressed. The template also identifies the result space to receive the decompressed data.

The *decompress data template* must be aligned on a 16-byte boundary. The format is as follows:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Reserved (binary 0)	Char(4)
4	4	Result area length	Bin(4)
8	8	Actual result length	Bin(4) +
12	C	Reserved (binary 0)	Char(20)
32	20	Source space pointer	Space pointer
48	30	Result space pointer	Space pointer
64	40	—- End —-	

Note: The input value associated with template fields annotated with a plus sign (+) are ignored by the instruction; these fields are updated by the instruction to return information about instruction execution.

The data at the location specified by the **source space pointer** is decompressed and stored at the location specified by the **result space pointer**. The **actual result length** is set to the number of bytes in the decompressed result. The source data is not modified.

The **result area length** field value must be greater than or equal to zero. A zero value means not specified. The length of the source data is not supplied in the template because this length is contained within the compressed data.

If the decompressed result data will not fit in the result area (as specified by the *result area length*), the decompression is stopped and only as many decompressed bytes as will fit in the result area are stored. The *actual result length* is always set to the full length of the result, which may be larger than the *result area length*.

The compressed data (previously compressed with CPRDATA) contains a **signature** which is checked by DCPDATA. The *signature* indicates which compression algorithm was used to compress the data. If the *signature* is invalid, an *invalid compressed data* (hex 0C14) exception is signaled. It is possible that the *signature* appears valid even though the compressed data has been corrupted. In almost all cases, the DCPDATA instruction will signal the *invalid compressed data* (hex 0C14) exception. Data corruption will not be detected only in the case when the decompression algorithm applied to the corrupted data produces the correct number of decompressed bytes.

It is not possible to corrupt the compressed data in such a way that the DCPDATA instruction would fail (that is, function check) or fail to terminate (that is, loop).

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C14 Invalid Compressed Data

10 Damage Encountered

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Decrement Date (DECD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0414	Result date	Source date	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

Bound program access

```
Built-in number for DECD is 96.
DECD (
      result_date            : address
      source_date            : address
      duration               : address of packed decimal
      instruction_template   : address
)
```

Description: The date specified by operand 2 is decremented by the date *duration* specified by operand 3. The resulting date is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the *instruction template*.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Instruction template	Char(*)	
0	0		Instruction template size	Bin(4)
4	4		Operand 1 data definitional attribute template number	UBin(2)
6	6		Operand 2 data definitional attribute template number	UBin(2)
8	8		Operand 3 data definitional attribute template number	UBin(2)
10	A		Operand 1 length	UBin(2)
12	C		Operand 2 length	UBin(2)
14	E		Operand 3 length	UBin(2)
14	E		Fractional number of digits	Char(1)
15	F		Total number of digits	Char(1)
16	10		Input indicators	Char(2)
16	10		End of month adjustment	Bit 0

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = No adjustment	
			1 = Adjustment	
16	10		Tolerate data decimal errors	Bit 1
			0 = No toleration	
			1 = Tolerate	
16	10		Reserved (binary 0)	Bits 2-15
18	12		Output indicators	Char(2)
18	12		End of month adjustment	Bit 0
			0 = No adjustment	
			1 = Adjustment	
18	12		Reserved (binary 0)	Bits 1-15
20	14		Reserved (binary 0)	Char(22)
42	2A		Data definitional attribute template list	Char(*)
42	2A		Size of the DDAT list	UBin(4)
46	2E		Number of DDATs	UBin(2)
48	30		Reserved (binary 0)	Char(10
58	3A		DDAT offset	[*] UBin(4)
*	*		Data definitional attribute template	[*] Char(*)
*	*	—- End —-		

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a date and must be identical. The DDAT for operand 3 must be valid for a date duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length, **operand 2 length**, and **operand 3 length** are specified in number of bytes.

The input indicator, **end of month adjustment**, is used to allow or disallow the occurrence of an end of month adjustment.

The input indicator, **tolerate decimal data errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The output indicator, **end of month adjustment**, is used to indicate an end of month adjustment, when end of month adjustments are allowed.

End of month adjustment is the following concept. For SAA(R), the result of subtracting a 1 month duration from the Gregorian date 03/31/1989 is 02/28/1989. The days portion is adjusted to fit the month, 31 is changed to 28. When this happens, the *end of month adjustment* output indicator is set to *adjustment*.

When *end of month adjustments are not allowed*, the month and year definitions in the data definition attribute template must have values greater than zero, otherwise a *template value invalid* (hex 3801) exception will be signalled. The result of subtracting a 1 month duration from the Gregorian date 03/31/1989 is 03/01/1989, when the definition of a month is 30 days.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes definitional attributes of the operands. The length of the date and date duration character operands will be defined by the template. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C15 Date Boundary Overflow

0C16 Data Format Error

0C17 Data Value Error

0C18 Date Boundary Underflow

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Decrement Time (DECT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0444	Result time	Source time	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

Bound program access

```
Built-in number for DECT is 98.
DECT (
      result_time            : address
      source_time            : address
      duration               : address of packed decimal
      instruction_template   : address
)
```

Description: The time specified by operand 2 is decremented by the time *duration* specified by operand 3. The resulting time is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the *instruction template*.

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Instruction template	Char(*)		
0	0		Instruction template size	Bin(4)	
4	4		Operand 1 data definitional attribute template number	UBin(2)	
6	6		Operand 2 data definitional attribute template number	UBin(2)	
8	8		Operand 3 data definitional attribute template number	UBin(2)	
10	A		Operand 1 length	UBin(2)	
12	C		Operand 2 length	UBin(2)	
14	E		Operand 3 length	UBin(2)	
14	E		Fractional number of digits		Char(1)
15	F		Total number of digits		Char(1)
16	10		Input indicators	Char(2)	
16	10		Reserved (binary 0)		Bit 0
16	10		Tolerate data decimal errors 0 = No toleration 1 = Tolerate		Bit 1
16	10		Reserved (binary 0)		Bits 2-15
18	12		Reserved (binary 0)	Char(24)	
42	2A		Data definitional attribute template list	Char(*)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
42	2A		Size of the DDAT list	UBin(4
46	2E		Number of DDATs	UBin(2
48	30		Reserved (binary 0)	Char(1
58	3A		DDAT offset	[*] UBin(4
*	*		Data definitional attribute template	[*] Char(*)
*	*	—- End —-		

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a time and must be identical. The DDAT for operand 3 must be valid for a time duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length, **operand 2 length**, and **operand 3 length** are specified in number of bytes.

The input indicator, **tolerate data decimal errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the time and time duration character operands will be defined by the templates. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

0C Computation

- 0C02 Decimal Data
- 0C16 Data Format Error
- 0C17 Data Value Error

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Decrement Timestamp (DECTS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
042C	Result timestamp	Source timestamp	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

Bound program access

```
Built-in number for DECTS is 100.
DECTS (
       result_timestamp       : address
       source_timestamp       : address
       duration               : address of packed decimal
       instruction_template   : address
)
```

Description: The timestamp specified by operand 2 is decremented by the date, time, or timestamp *duration* specified by operand 3. The resulting timestamp is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the *instruction template*.

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Instruction template	Char(*)	
0	0		Instruction template size	Bin(4)
4	4		Operand 1 data definitional attribute template number	UBin(2)
6	6		Operand 2 data definitional attribute template number	UBin(2)

Offset Dec	Hex	Field Name	Data Type and Length	
8	8		Operand 3 data definitional attribute template number	UBin(2)
10	A		Operand 1 length	UBin(2)
12	C		Operand 2 length	UBin(2)
14	E		Operand 3 length	UBin(2)
14	E		Fractional number of digits	Char(1)
15	F		Total number of digits	Char(1)
16	10		Input indicators	Char(2)
16	10		End of month adjustment	Bit 0
			0 = No adjustment	
			1 = Adjustment	
16	10		Tolerate data decimal errors	Bit 1
			0 = No toleration	
			1 = Tolerate	
16	10		Reserved (binary 0)	Bits 2-15
18	12		Output indicators	Char(2)
18	12		End of month adjustment	Bit 0
			0 = No adjustment	
			1 = Adjustment	
18	12		Reserved (binary 0)	Bits 1-15
20	14		Reserved (binary 0)	Char(22)
42	2A		Data definitional attribute template list	Char(*)
42	2A		Size of the DDAT list	UBin(4)
46	2E		Number of DDATs	UBin(2)
48	30		Reserved (binary 0)	Char(10)
58	3A		DDAT offset	[*] UBin(4)
*	*		Data definitional attribute template	[*] Char(*)
*	*	— End —		

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a timestamp and identical. The DDAT for operand 3 must be valid for a timestamp duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length, **operand 2 length**, and **operand 3 length** are specified in number of bytes.

The input indicator, **end of month adjustment**, is used to allow or disallow the occurrence of an end of month adjustment.

The input indicator, **tolerate data decimal errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.

2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The output indicator, **end of month adjustment**, is used to indicate an end of month adjustment, when end of month adjustments are allowed.

End of month adjustment is the following concept. For SAA(R), the result of subtracting a 1 month duration from the date 03/31/1989 is 02/28/1989. The days portion is adjusted to fit the month, 31 is changed to 28. When this happens, the *end of month adjustment* output indicator is set to *adjustment*.

When *end of month adjustments are not allowed*, the month and year definitions in the data definition attribute template must have values greater than zero, otherwise a *template value invalid* (hex 3801) exception will be signaled. The result of subtracting a 1 month duration from the Gregorian date 03/31/1989 is 03/01/1989, when the definition of a month is 30 days.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the timestamp and duration character operands will be defined by the template. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

- 0C02 Decimal Data
- 0C15 Date Boundary Overflow
- 0C16 Data Format Error
- 0C17 Data Value Error
- 0C18 Date Boundary Underflow

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

- 3202 Scalar Attributes Invalid
- 3203 Scalar Value Invalid

36 Space Management

- 3601 Space Extension/Truncation

38 Template Specification

- 3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Dequeue (DEQ)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4-5
DEQ 1033		Message prefix	Message text	Queue or queue template	
DEQB 1C33	Branch options	Message prefix	Message text	Queue or queue template	Branch targets
DEQI 1833	Indicator options	Message prefix	Message text	Queue or queue template	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Space pointer.

Operand 3: System pointer or space pointer data object.

Operand 4-5:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for DEQ is 41.
DEQ (
     message_prefix   : address
     message_text     : address
     queue            : address of system pointer OR
                        address of space pointer(16)
) :   signed binary(4) /* return_code */
```

The return code is set as follows:

Return code	Meaning
1	Message Dequeued.
0	Message Not Dequeued.

This built-in function corresponds to the branch and indicator forms of the DEQ operation.

-- OR --

```
Built-in number for DEQWAIT is 42.
DEQWAIT (
         message_prefix   : address
         message_text     : address
         queue            : address of system pointer OR
                            address of space pointer(16)
)
```

This built-in function corresponds to the non-branch and non-indicator form of the DEQ operation.

Description: Retrieves a queue message based on the *queue type* (FIFO, LIFO, or keyed) specified during the queue's creation. If the queue was created with the *keyed* option, messages can be retrieved by any of the following relationships between an enqueued *message key* and a *selection key* specified in operand 1 of the Dequeue instruction: =, <>, >, <, <=, and >=. If the queue was created with either the *LIFO* or *FIFO* attribute, then only the next message can be retrieved from the queue.

If a message is not found that satisfies the dequeue selection criterion and the branch or indicator options are not specified, the thread waits until a message arrives to satisfy the dequeue or until the *dequeue wait time-out* expires. If branch or indicator options are specified, the thread is not placed in the dequeue wait state and either the control flow is altered according to the branch options, or indicator values are set based on the presence or absence of a message to be dequeued.

If operand 3 is a system pointer, the message is dequeued from the *queue* specified by operand 3. If operand 3 is a space pointer, the message is dequeued from the queue which is specified in the template pointed to by the space pointer. The format of this template is given later in this section. The criteria for message selection are given in the *message prefix* specified by operand 1. The *message text* is returned in the space specified by operand 2, and the *message prefix* is returned in the scalar specified by operand 1. If an exception is signaled, the *message text* and *message prefix* may be changed, but do not contain valid data. Improper alignment results in an exception being signaled. The format of the *message prefix* is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Timestamp of enqueue of message	Char(8) ++	
8	8	Dequeue wait time-out value (ignored if branch options specified)	Char(8) +	
16	10	Size of message dequeued (The maximum allowable size of a queue message is 64 K bytes.)	Bin(4) ++	
20	14	Access state modification option indicator and message selection criteria	Char(1) +	
20	14		Access state modification option when entering Dequeue wait	Bit 0 +
			0 = Access state is not modified	
			1 = Access state is modified	
20	14		Access state modification option when leaving Dequeue wait	Bit 1 +
			0 = Access state is not modified	
			1 = Access state is modified	
20	14		Multiprogramming level option	Bit 2 +
			0 = Leave current MPL set at Dequeue wait	
			1 = Remain in current MPL set at Dequeue wait	
20	14		Time-out option	Bit 3 +
			0 = Wait for specified time, then signal time-out exception	
			1 = Wait indefinitely	
20	14		Actual key to input key relationship (for keyed queue)	Bits 4-7 +
			0010 = Greater than	
			0100 = Less than	
			0110 = Not equal	
			1000 = Equal	
			1010 = Greater than or equal	
			1100 = Less than or equal	
21	15	Search key (ignored for FIFO/LIFO queues but must be present for FIFO/LIFO queues with nonzero key length values)	Char(key length) +	
*	*	Message key	Char(key length) ++	
*	*	—- End —-		

Note: Fields shown here with one plus sign (+) indicate input to the instruction, and fields shown here with two plus signs (++) are returned by the machine.

A nonzero **dequeue wait time-out value** overrides any *dequeue wait time-out* value specified as the current process attribute. A zero *dequeue wait time-out value* causes the wait time-out value to be taken from the current process attribute. If all wait time-out values are 0 (from the Dequeue instruction and the current process attribute), a *dequeue time-out* (hex 3A01) exception is signaled. See "Standard Time Format" on page 1272 for additional information on the format of the *dequeue wait time-out*. The maximum *dequeue*

wait time-out interval allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum wait time-out causes the maximum wait time-out to be used.

The **size of message dequeued** is returned in the *message prefix*. The *size of message dequeued* can be less than or equal to the *maximum size of message* specified when the queue was created. When dequeuing from a keyed queue, the length of the *search key* field and the length of the *message key* field (in the message key prefix specified in operand 1) are determined implicitly by the attributes of the queue being accessed. If the message text on the queue contains pointers, the *message text* operand must be 16-byte aligned.

The **access state** of the process access group is modified when a Dequeue instruction results in a wait and the following conditions exist:

•
- The process' *instruction wait initiation access state* control attribute specifies allow access state modification
- The *dequeue access state modification* option specifies *modify access state*
- The **multiprogramming level option** specifies *leave MPL* set during wait.
- The process is not multi-threaded (i.e. the waiting thread is the only thread in the process)

The thread will remain in the current MPL set for an implementation-defined period which will not exceed 2 seconds, if the *multiprogramming level option* specifies *remain in current MPL set at Dequeue wait*. If the wait has not been satisfied at the end of this period, the thread will automatically be removed from the current MPL set. The automatic removal does not change or affect the total wait time specified for the thread by the *dequeue wait time-out value*.

Operand 3 can be a system pointer or a space pointer. If it is a system pointer, this pointer will be addressing the queue from which the message is to be dequeued. If it is a space pointer, this pointer will be addressing a template which will contain the system pointer to the queue as well as the *dequeue template extension*. The *queue template* is 32 bytes in length and must be aligned on a 16-byte boundary with the format as follows:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Queue	System pointer
16	10	Dequeue template extension	Char(16)
16	10		Extension options
16	10		Modify thread event mask option
			0 = Do not modify thread event mask
			1 = Modify thread event mask
16	10		Asynchronous signals processing option
			0 = Do not allow asynchronous signal processing
			1 = Allow asynchronous signal processing during
16	10		Reserved (binary 0)
17	11		Extension area
17	11		New thread event mask
19	13		Previous thread event mask
21	15		Reserved (binary 0)
32	20	— End —	

Note: Fields shown here with one plus sign (+) indicate input to the instruction, and fields shown here with two plus signs (++) are returned by the machine.

The **modify thread event mask option** controls the state of the event mask in the thread executing this instruction. If the *modify thread event mask option* field specifies to *modify thread event mask*, the thread event mask will be changed as specified by the **new thread event mask** field. When the thread event mask is changed, the current thread event mask will be returned in the **previous thread event mask** field. The *previous thread event mask* is only returned when the *modify thread event mask option* is set to 1.

If the system security level machine attribute is hex 40 or greater and the thread is running in user state, a *template value invalid* (hex 3801) exception is signalled if the *modify thread event mask option* is set to *modify thread event mask*.

If the *thread event mask* is in the *masked* state, the machine does not schedule signaled event monitors in the thread. The event monitors continue to be signaled by the machine or other threads. When the thread is modified to the *unmasked* state, event handlers are scheduled to handle those events that occurred while the thread was masked and those events occurring while in the unmasked state. The number of signals retained while the thread is masked is specified by the attributes of the event monitor associated with the process or thread.

The thread is automatically masked by the machine when event handlers are invoked. If the thread is unmasked in the event handler, other events can be handled if another enabled event monitor within that thread is signaled. If the thread is masked when it exits from the event handler, the machine explicitly unmasks the thread.

Valid masking values are:

0	Masked
256	Unmasked

Other values are reserved and must not be specified. If any other values are specified, a *template value invalid* (hex 3801) exception is signaled.

Whether masking or unmasking the current thread, the new mask takes effect upon completion of a satisfied dequeue.

The **asynchronous signals processing option** controls the action to be taken if an asynchronous signal is pending or received while in a Dequeue wait. If an asynchronous signal that is not blocked or ignored is generated for the process and the *asynchronous signals processing option* indicates *allow asynchronous signal processing during Dequeue wait*, the Dequeue wait will be terminated and an *asynchronous signal terminated MI wait* (hex 4C01) exception is signaled. Otherwise, when the *asynchronous signals processing option* indicates *do not allow asynchronous signal processing during Dequeue wait*, the process remains in the wait until a message arrives to satisfy the dequeue or until the *dequeue wait time-out value* expires.

Warning: Temporary Level 3 Header

Resultant Conditions

•

- Equal - message dequeued
- Not equal - message not dequeued

Authorization Required

•

- Retrieve
  –

  - Operand 3
- Execute
  -
  - Contexts referenced for address resolution

Lock Enforcement

-
- Materialize
  -
  - Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C0E IASP Resources Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

30 Journal

3002 Entry Not Journaled

32 Scalar Specification

3203 Scalar Value Invalid

38 Template Specification

3801 Template Value Invalid

3A Wait Time-Out

3A01 Dequeue Time-Out

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

4C Signals Management

4C01 Asynchronous Signal Terminated MI Wait

Dequeue (DEQ)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4-5
DEQ 1033		Message prefix	Message text	Queue or queue template	
DEQB 1C33	Branch options	Message prefix	Message text	Queue or queue template	Branch targets
DEQI 1833	Indicator options	Message prefix	Message text	Queue or queue template	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Space pointer.

Operand 3: System pointer or space pointer data object.

Operand 4-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for DEQ is 41.
DEQ (
     message_prefix   : address
     message_text     : address
     queue            : address of system pointer OR
                        address of space pointer(16)
) :   signed binary(4) /* return_code */
```

The return code is set as follows:

Return code

Meaning

1

Message Dequeued.

0

Message Not Dequeued.

This built-in function corresponds to the branch and indicator forms of the DEQ operation.

-- OR --

```
Built-in number for DEQWAIT is 42.
DEQWAIT (
         message_prefix   : address
         message_text     : address
         queue            : address of system pointer OR
                            address of space pointer(16)
)
```

This built-in function corresponds to the non-branch and non-indicator form of the DEQ operation.

Description: Retrieves a queue message based on the *queue type* (FIFO, LIFO, or keyed) specified during the queue's creation. If the queue was created with the *keyed* option, messages can be retrieved by any of the following relationships between an enqueued *message key* and a *selection key* specified in operand 1 of the Dequeue instruction: =, <>, >, <, <=, and >=. If the queue was created with either the *LIFO* or *FIFO* attribute, then only the next message can be retrieved from the queue.

If a message is not found that satisfies the dequeue selection criterion and the branch or indicator options are not specified, the thread waits until a message arrives to satisfy the dequeue or until the *dequeue wait time-out* expires. If branch or indicator options are specified, the thread is not placed in the dequeue wait state and either the control flow is altered according to the branch options, or indicator values are set based on the presence or absence of a message to be dequeued.

If operand 3 is a system pointer, the message is dequeued from the *queue* specified by operand 3. If operand 3 is a space pointer, the message is dequeued from the queue which is specified in the template pointed to by the space pointer. The format of this template is given later in this section. The criteria for message selection are given in the *message prefix* specified by operand 1. The *message text* is returned in the space specified by operand 2, and the *message prefix* is returned in the scalar specified by operand 1. If an exception is signaled, the *message text* and *message prefix* may be changed, but do not contain valid data. Improper alignment results in an exception being signaled. The format of the *message prefix* is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Timestamp of enqueue of message	Char(8) ++	
8	8	Dequeue wait time-out value (ignored if branch options specified)	Char(8) +	
16	10	Size of message dequeued (The maximum allowable size of a queue message is 64 K bytes.)	Bin(4) ++	
20	14	Access state modification option indicator and message selection criteria	Char(1) +	
20	14		Access state modification option when entering Dequeue wait	Bit 0 +
			0 = Access state is not modified	
			1 = Access state is modified	
20	14		Access state modification option when leaving Dequeue wait	Bit 1 +
			0 = Access state is not modified	
			1 = Access state is modified	
20	14		Multiprogramming level option	Bit 2 +
			0 = Leave current MPL set at Dequeue wait	
			1 = Remain in current MPL set at Dequeue wait	
20	14		Time-out option	Bit 3 +
			0 = Wait for specified time, then signal time-out exception	
			1 = Wait indefinitely	
20	14		Actual key to input key relationship	Bits 4-7 +

Offset Dec	Offset Hex	Field Name	Data Type and Length
			(for keyed queue) **0010 =** Greater than **0100 =** Less than **0110 =** Not equal **1000 =** Equal **1010 =** Greater than or equal **1100 =** Less than or equal
21	15	Search key (ignored for FIFO/LIFO queues but must be present for FIFO/LIFO queues with nonzero key length values)	Char(key length) +
*	*	Message key	Char(key length) ++
*	*	—- End —-	

Note: Fields shown here with one plus sign (+) indicate input to the instruction, and fields shown here with two plus signs (++) are returned by the machine.

A nonzero **dequeue wait time-out value** overrides any *dequeue wait time-out* value specified as the current process attribute. A zero *dequeue wait time-out value* causes the wait time-out value to be taken from the current process attribute. If all wait time-out values are 0 (from the Dequeue instruction and the current process attribute), a *dequeue time-out* (hex 3A01) exception is signaled. See "Standard Time Format" on page 1272 for additional information on the format of the *dequeue wait time-out*. The maximum *dequeue wait time-out* interval allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum wait time-out causes the maximum wait time-out to be used.

The **size of message dequeued** is returned in the *message prefix*. The *size of message dequeued* can be less than or equal to the *maximum size of message* specified when the queue was created. When dequeuing from a keyed queue, the length of the *search key* field and the length of the *message key* field (in the message key prefix specified in operand 1) are determined implicitly by the attributes of the queue being accessed. If the message text on the queue contains pointers, the *message text* operand must be 16-byte aligned.

The **access state** of the process access group is modified when a Dequeue instruction results in a wait and the following conditions exist:

-
- The process' *instruction wait initiation access state* control attribute specifies allow access state modification
- The *dequeue access state modification* option specifies *modify access state*
- The **multiprogramming level option** specifies *leave MPL* set during wait.
- The process is not multi-threaded (i.e. the waiting thread is the only thread in the process)

The thread will remain in the current MPL set for an implementation-defined period which will not exceed 2 seconds, if the *multiprogramming level option* specifies *remain in current MPL set at Dequeue wait*. If the wait has not been satisfied at the end of this period, the thread will automatically be removed from the current MPL set. The automatic removal does not change or affect the total wait time specified for the thread by the *dequeue wait time-out value*.

Operand 3 can be a system pointer or a space pointer. If it is a system pointer, this pointer will be addressing the queue from which the message is to be dequeued. If it is a space pointer, this pointer will

be addressing a template which will contain the system pointer to the queue as well as the *dequeue template extension*. The *queue template* is 32 bytes in length and must be aligned on a 16-byte boundary with the format as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Queue	System pointer
16	10	Dequeue template extension	Char(16)
16	10		Extension options
16	10		Modify thread event mask option
			0 = Do not modify thread event mask
			1 = Modify thread event mask
16	10		Asynchronous signals processing option
			0 = Do not allow asynchronous signal proces
			1 = Allow asynchronous signal processing du
16	10		Reserved (binary 0)
17	11		Extension area
17	11		New thread event mask
19	13		Previous thread event mask
21	15		Reserved (binary 0)
32	20	—- End —-	

Note: Fields shown here with one plus sign (+) indicate input to the instruction, and fields shown here with two plus signs (++) are returned by the machine.

The **modify thread event mask option** controls the state of the event mask in the thread executing this instruction. If the *modify thread event mask option* field specifies to *modify thread event mask*, the thread event mask will be changed as specified by the **new thread event mask** field. When the thread event mask is changed, the current thread event mask will be returned in the **previous thread event mask** field. The *previous thread event mask* is only returned when the *modify thread event mask option* is set to 1.

If the system security level machine attribute is hex 40 or greater and the thread is running in user state, a *template value invalid* (hex 3801) exception is signalled if the *modify thread event mask option* is set to *modify thread event mask*.

If the *thread event mask* is in the *masked* state, the machine does not schedule signaled event monitors in the thread. The event monitors continue to be signaled by the machine or other threads. When the thread is modified to the *unmasked* state, event handlers are scheduled to handle those events that occurred while the thread was masked and those events occurring while in the unmasked state. The number of signals retained while the thread is masked is specified by the attributes of the event monitor associated with the process or thread.

The thread is automatically masked by the machine when event handlers are invoked. If the thread is unmasked in the event handler, other events can be handled if another enabled event monitor within that thread is signaled. If the thread is masked when it exits from the event handler, the machine explicitly unmasks the thread.

Valid masking values are:

0	Masked
256	Unmasked

Other values are reserved and must not be specified. If any other values are specified, a *template value invalid* (hex 3801) exception is signaled.

Whether masking or unmasking the current thread, the new mask takes effect upon completion of a satisfied dequeue.

The **asynchronous signals processing option** controls the action to be taken if an asynchronous signal is pending or received while in a Dequeue wait. If an asynchronous signal that is not blocked or ignored is generated for the process and the *asynchronous signals processing option* indicates *allow asynchronous signal processing during Dequeue wait*, the Dequeue wait will be terminated and an *asynchronous signal terminated MI wait* (hex 4C01) exception is signaled. Otherwise, when the *asynchronous signals processing option* indicates *do not allow asynchronous signal processing during Dequeue wait*, the process remains in the wait until a message arrives to satisfy the dequeue or until the *dequeue wait time-out value* expires.

Warning: Temporary Level 3 Header

Resultant Conditions

•

- Equal - message dequeued
- Not equal - message not dequeued

Authorization Required

•

- Retrieve
  - –
  - – Operand 3
- Execute
  - –
  - – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  - –
  - – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

- 1004 System Object Damage State
- 1005 Authority Verification Terminated Due to Damaged Object
- 1044 Partial System Object Damage

1A Lock State

- 1A01 Invalid Lock State

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded
- 1C0E IASP Resources Exceeded
- 1C11 Independent ASP Varied Off

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2207 Authority Verification Terminated Due to Destroyed Object
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid
- 2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

30 Journal

- 3002 Entry Not Journaled

32 Scalar Specification

3203 Scalar Value Invalid

38 Template Specification

3801 Template Value Invalid

3A Wait Time-Out

3A01 Dequeue Time-Out

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

4C Signals Management

4C01 Asynchronous Signal Terminated MI Wait

Destroy Activation Group-Based Heap Space (DESHS)

Op Code (Hex)	Operand 1
03B1	Heap identifier

Operand 1: Binary(4) variable scalar.

Bound program access

```
Built-in number for DESHS is 113.
DESHS (
       heap_identifier   : address of signed binary(4) OR
                           address of unsigned binary(4)
)
```

Note: The term ″heap space″ in this instruction refers to an ″activation group-based heap space″.

Description: This instruction destroys and removes from the current activation group the heap space specified by the *heap identifier* in operand 1. Subsequent use of this *heap identifier* within the activation group will result in an *invalid heap identifier* (hex 4501) exception. The *heap identifier* was returned on the Create Activation Group-Based Heap Space (CRTHS) instruction. An attempt to destroy the default heap space (heap identifier value of 0) will result in an *invalid request* (hex 4502) exception.

Space pointer references to heap space allocations from a destroyed heap space will cause unpredictable results.

All heap spaces are implicitly destroyed when the activation group in which they were created is destroyed.

Operand 1 is not modified by the instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

45 Heap Space

4501 Invalid Heap Identifier

4502 Invalid Request

4505 Heap Space Destroyed

4506 Invalid Heap Space Condition

Destroy Independent Index (DESINX)

Op Code (Hex)	Operand 1
0451	Index

Operand 1: System pointer.

Bound program access

```
Built-in number for DESINX is 35.
DESINX (
        index   : address of system pointer
)
```

Description: A previously created *index* identified by operand 1 is destroyed, and addressability to the object is removed from any context in which addressability exists. The system pointer identified by operand 1 is not modified by the instruction, and a subsequent reference to the destroyed index through the pointer results in an *object destroyed* (hex 2202) exception.

Permanent index objects cannot be destroyed by user state programs when the system security level is 40 or above.

Warning: Temporary Level 3 Header

Authorization Required

•

- Object control
  –
  – Operand 1
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Contexts referenced for address resolution
- Object control
  –
  – Operand 1
- Modify
  –
  – Access group which contains operand 1

- Context which addresses operand 1
- User profile which owns index

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Destroy Pointer-Based Mutex (DESMTX)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03C7	Mutex	Mutex destroy options	Result

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Signed binary(4) variable scalar.

Bound program access

```
Built-in number for DESMTX is 162.
DESMTX (
        mutex                   : address
        mutex_destroy_options   : address of unsigned binary(4) value
) :   signed binary(4)  /* result */
```

Note: The term ″mutex″ in this instruction refers to a ″pointer-based mutex″.

Description: The *mutex* whose address is referenced by operand 1 is destroyed. The *mutex* is set to binary zero. All threads currently in the mutex wait state for this *mutex* are removed from the wait state and an EDESTROYED error number result is returned to each waiting thread. The ETYPE error is returned when the *mutex* operand references a synchronization object that is not a pointer-based mutex. The ENOTSUP error is returned when an attempt is made to destroy the mutex using a *mutex* operand that is a copy of the original mutex. See the CRTMTX instruction for additional information regarding mutex copies.

The space pointed to by operand 2 contains the *mutex destroy options*.

Result is used to indicate the success or failure of the DESMTX instruction.

Mutexes are temporary entities that do not persist beyond the current IPL in which they are created. Following a subsequent IPL, mutexes must be re-created before they can be used. Similarly, mutexes created in an independent ASP do not persist beyond the current vary on of the independent ASP in which they are created. Following a subsequent vary on, mutexes must be re-created before they can be used. Mutexes can be explicitly destroyed prior to a subsequent IPL or independent ASP vary off/vary on cycle by using the DESMTX instruction.

It is important to destroy mutexes when they are no longer needed. When a mutex is created, system resources are allocated for the mutex. These resources remain allocated until the mutex is destroyed or the system is IPLed, thereby leaving fewer mutex resources available in the system for other threads to use. In addition, performance degradation can occur as unused mutexes accumulate on the system and are not destroyed.

The *mutex* must be aligned on a 16-byte boundary.

The *mutex destroy options* referenced by operand 2 can have the following values:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Mutex destroy options	UBin(4)
		0 = Destroy mutex (this is the default option)	
4	4	—- End —-	

The **mutex destroy options** should be set to 0 in order to destroy a mutex. The *mutex* will not be destroyed if it is locked by another thread. An attempt to destroy a mutex when another thread has it locked will result in an EBUSY error number result.

All other values for *mutex destroy options* are reserved and will cause an EINVAL error number result to be returned.

If operand 2 in a bound program is a null pointer value, the default *mutex destroy options* are used. Operand 2 in non-bound programs must be a pointer to *mutex destroy options*. The *pointer does not exist* (hex 2401) exception is signaled if a null pointer value is used for operand 2 in a non-bound program.

If the *mutex* is destroyed by this instruction, then *result* is set to 0. If an error occurs, then the *result* is set to an error condition. The EINVAL error number is returned when an invalid operand is specified. The EPERM error number is returned when the address passed in operand 1 is in teraspace and the issuing thread does not have teraspace write permissions to that address.

The *mutex* must have been previously created by the CRTMTX instruction. Attempting to destroy a mutex that was not created or whose contents have been altered will cause one of the *pointer specification* exceptions to occur.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Error conditions

The *result* is set to one of the following:

EBUSY 3029 - Resource busy.

EINVAL 3021 - The value specified for the argument is not correct.

ENOTSUP 3440 - Operation not supported.

EPERM 3027 - Operation not permitted.

ETYPE 3493 - Object type mismatch.

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Destroy Space (DESS)

Op Code (Hex)	Operand 1
0025	Space to be destroyed

Operand 1: System pointer.

Bound program access
Built-in number for DESS is 26. DESS (space_to_be_destroyed : address of system pointer)

Description: The designated space is destroyed, and addressability to the space is deleted from a context if it is currently addressing the object. The pointer identified by operand 1 is not modified by the instruction, and a subsequent reference to the pointer causes an *object destroyed* (hex 2202) exception.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

–

– Contexts referenced for address resolution

• Object control

–

– Operand 1

Lock Enforcement

•

• Modify

–

– User profile owning object

  - Context addressing object
  - Access group containing object
- Object control
  -
  - Operand 1

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Divide (DIV)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
DIV 104F		Quotient	Dividend	Divisor	
DIVR 124F		Quotient	Dividend	Divisor	
DIVI 184F	Indicator options	Quotient	Dividend	Divisor	Indicator targets
DIVIR 1A4F	Indicator options	Quotient	Dividend	Divisor	Indicator targets
DIVB 1C4F	Branch options	Quotient	Dividend	Divisor	Branch targets
DIVBR 1E4F	Branch options	Quotient	Dividend	Divisor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4-7:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
DIVS 114F		Quotient/Dividend	Divisor	
DIVSR 134F		Quotient/Dividend	Divisor	
DIVIS 194F	Indicator options	Quotient/Dividend	Divisor	Indicator targets
DIVISR 1B4F	Indicator options	Quotient/Dividend	Divisor	Indicator targets
DIVBS 1D4F	Branch options	Quotient/Dividend	Divisor	Branch targets
DIVBSR 1F4F	Branch options	Quotient/Dividend	Divisor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The *quotient* is the result of dividing the *dividend* by the *divisor.*

Operands can have floating-point, packed or zoned decimal, signed or unsigned binary type.

Source operands are the *dividend* and *divisor.* The receiver operand is the *quotient.*

If operands are not of the same type, source operands are converted according to the following rules:

1. If any one of the operands has floating point type, source operands are converted to floating point type.
2. Otherwise, if any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.
3. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Source operands are divided according to their type. Floating point operands are divided using floating point division. Packed decimal operands are divided using packed decimal division. Unsigned binary division is used with unsigned source operands. Signed binary operands are divided using two's complement binary division.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary division execute faster than either packed decimal or floating point division.

All of the operands must be numeric with any implicit conversions occurring according to the rules of arithmetic operations as outlined in Arithmetic Operations.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

If the *divisor* has a numeric value of zero, a *zero divide* (hex 0C0B) exception or *floating-point zero divide* (hex 0C0E) exception is signaled respectively for fixed-point versus floating-point operations. If the dividend has a value of zero, the result of the division is a zero quotient value.

For a decimal operation, the precision of the result of the divide operation is determined by the number of fractional digit positions specified for the quotient. In other words, the divide operation will be performed so as to calculate a resultant quotient of the same precision as that specified for the *quotient* operand. If necessary, internal alignment of the assumed decimal point for the *dividend* and *divisor* operands is performed to ensure the correct precision for the resultant quotient value. These internal alignments are not subject to detection of the decimal point alignment exception. An internal quotient value will be calculated for any combination of decimal attributes which may be specified for the instruction's operands. However, the assignment of the result to the *quotient* operand is subject to detection of the *size* (hex 0C0A) exception thereby limiting the assignment to, at most, the rightmost 31 digits of the calculated result.

Floating-point division uses exponent subtraction and significand division.

If the *dividend* operand is shorter than the *divisor* operand, it is logically adjusted to the length of the *divisor* operand.

For fixed-point computations and for the significand division of a floating-point computation, the division operation is performed according to the rules of algebra. Unsigned binary is treated as a positive number for the algebra.

For a floating-point computation, the operation is performed as if to infinite precision.

The result of the operation is copied into the *quotient* operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the *quotient* operand, aligned at the assumed decimal point of the *quotient* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules for arithmetic operations as outlined in Arithmetic Operations. If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

For the optional round form of the instruction, specification of a floating-point receiver operand is invalid.

For fixed-point operations in programs that request to be notified of *size* (hex 0C0A) exceptions, if nonzero digits are truncated from the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For floating-point operations that involve a fixed-point receiver field, if nonzero digits would be truncated from the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point *quotient* operand, if the exponent of the resultant value is either too large or too small to be represented in the quotient field, the *floating-point overflow* (hex 0C06) exception and *floating-point underflow* (hex 0C07) exception are signaled, respectively.

Resultant Conditions:

•

- Positive-The algebraic value of the numeric scalar *quotient* is positive.
- Negative-The algebraic value of the numeric scalar *quotient* is negative.
- Zero-The algebraic value of the numeric scalar *quotient* is zero.
- Unordered-The value assigned a floating-point *quotient* operand is NaN.

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0B Zero Divide

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Divide with Remainder (DIVREM)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-7]
DIVREM 1074		Quotient	Dividend	Divisor	Remainder	
DIVREMR 1274		Quotient	Dividend	Divisor	Remainder	
DIVREMI 1874	Indicator options	Quotient	Dividend	Divisor	Remainder	Indicator targets
DIVREMIR 1A74	Indicator options	Quotient	Dividend	Divisor	Remainder	Indicator targets
DIVREMB 1C74	Branch options	Quotient	Dividend	Divisor	Remainder	Branch targets

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-7]
DIVREMBR 1E74	Branch options	Quotient	Dividend	Divisor	Remainder	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4: Numeric variable scalar.

Operand 5-7:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
DIVREMS 1174		Quotient/Dividend	Divisor	Remainder	
DIVREMSR 1374		Quotient/Dividend	Divisor	Remainder	
DIVREMIS 1974	Indicator options	Quotient/Dividend	Divisor	Remainder	Indicator targets
DIVREMISR 1B74	Indicator options	Quotient/Dividend	Divisor	Remainder	Indicator targets
DIVREMBS 1D74	Branch options	Quotient/Dividend	Divisor	Remainder	Branch targets
DIVREMBSR 1F74	Branch options	Quotient/Dividend	Divisor	Remainder	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric variable scalar.

Operand 4-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The *quotient* is the result of dividing the *dividend* by the *divisor.* The *remainder* is the *dividend* minus the product of the *divisor* and *quotient.*

Operands can have packed or zoned decimal, signed or unsigned binary type.

Source operands are the *dividend* and *divisor.* The receiver operands are the *quotient* and *remainder.*

If operands are not of the same type, source operands are converted according to the following rules:

1. If any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.

2. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Source operands are divided according to their type. Packed decimal operands are divided using packed decimal division. Unsigned binary division is used with unsigned source operands. Signed binary operands are divided using two's complement binary division.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary division execute faster than packed decimal division.

The operands must be numeric with any implicit conversions occurring according to the rules for arithmetic operations as outlined in Arithmetic Operations.

Floating-point is not supported for this instruction.

If the *divisor* operand has a numeric value of 0, a *zero divide* (hex 0C0B) exception is signaled. If the *dividend* operand has a value of 0, the result of the division is a zero value *quotient* and *remainder.*

For a decimal operation, the precision of the result of the divide operation is determined by the number of fractional digit positions specified for the *quotient.* In other words, the divide operation will be performed so as to calculate a resultant quotient of the same precision as that specified for the *quotient* operand. If necessary, internal alignment of the assumed decimal point for the *dividend* and *divisor* operands is performed to ensure the correct precision for the resultant quotient value. These internal alignments are not subject to detection of the decimal point alignment exception. An internal quotient value will be calculated for any combination of decimal attributes which may be specified for the instruction's operands. However, the assignment of the result to the *quotient* operand is subject to detection of the *size* (hex 0C0A) exception thereby limiting the assignment to, at most, the rightmost 31 digits of the calculated result.

If the *dividend* operand is shorter than the *divisor* operand, it is logically adjusted to the length of the *divisor* operand.

The division operation is performed according to the rules of algebra. Unsigned binary is treated as a positive number for the algebra. The quotient result of the operation is copied into the *quotient* operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the *quotient* operand, aligned at the assumed decimal point of the *quotient* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations as outlined in Arithmetic Operations. If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

After the quotient numeric value has been determined, the numeric value of the *remainder* operand is calculated as follows:

```
Remainder = Dividend - (Quotient*Divisor)
```

If the optional round form of this instruction is being used, the rounding applies to the *quotient* but not the *remainder.* The quotient value used to calculate the remainder is the resultant value of the division. The resultant value of the calculation is copied into the *remainder* operand. The sign of the remainder is the same as that of the *dividend* operand unless the remainder has a value of 0, in which case its sign is positive. If the *remainder* operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the *remainder* operand, aligned at the assumed decimal point of the *remainder* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations as outlined in the Arithmetic Operations. If significant digits are truncated off the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled (in programs that request size exceptions to be signaled), the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Warning: Temporary Level 3 Header

Resultant Conditions

The algebraic value of the numeric scalar quotient is

•

- Positive
- Negative
- Zero

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C0A Size

0C0B Zero Divide

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Edit (EDIT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10E3	Receiver	Source	Edit mask

Operand 1: Character variable scalar or data-pointer-defined character scalar.

Operand 2: Numeric scalar or data-pointer-defined numeric scalar.

Operand 3: Character variable scalar or data-pointer-defined character scalar.

Bound program access

```
Built-in number for LBEDIT is 137.
LBEDIT (
        receiver            : address
        receiver_length     : address of unsigned binary(4)
        source              : address of signed binary(4) OR
                              address of unsigned binary(4) OR
                              address of packed decimal (1 to 63 digits) OR
                              address of zoned decimal (1 to 63 digits)
        source_attributes   : address
        mask                : address
        mask_length         : address of unsigned binary(4)
)
```

The *receiver*, *source* and *mask* parameters correspond to operands 1, 2 and 3 on the EDIT operation.

The *source_attributes* is a structure which describes the data attributes of the source value. The format of this structure matches that of the third operand on the CVTCN and CVTNC operations.

The *receiver_length* and *mask_length* parameters contain the length, in bytes, of the receiver and mask. They are expected to contain a value between 1 and 256.

-- OR --

```
EDITPD (
        receiver          : address
        receiver_length   : unsigned binary(4)
        source            : address of packed decimal (1 to 63 digits)
        source_length     : unsigned binary(4)
        mask              : address
        mask_length       : unsigned binary(4)
)
```

This built-in function supports the EDIT operation when the source contains a packed decimal value. The *receiver*, *source* and *mask* parameters correspond to operands 1, 2 and 3 on the EDIT operation.

The *source_length* parameter contains the length, in digits, of the source. It is expected to contain a value between 1 and 63.

The *receiver_length* and *mask_length* parameters contain the length, in bytes, of the receiver and mask. They are expected to contain a value between 1 and 256.

Description: The value of a numeric scalar is transformed from its internal form to character form suitable for display at a source/sink device. The following general editing functions can be performed during transforming of the *source* operand to the *receiver* operand:

-
- Unconditional insertion of a source value digit with a zone as a function of the source value's algebraic sign
- Unconditional insertion of a mask operand character string
- Conditional insertion of one of two possible *edit* mask operand character strings as a function of the source value's algebraic sign
- Conditional insertion of a source value digit or an *edit mask* operand replacement character as a function of source value leading zero suppression
- Conditional insertion of either an *edit mask* operand character string or a series of replacement characters as a function of source value leading zero suppression
- Conditional floating insertion of one of two possible *edit mask* operand character strings as a function of both the algebraic sign of the source value and leading zero suppression

The operation is performed by transforming the *source* (operand 2) under control of the *edit mask* (operand 3) and placing the result in the *receiver* (operand 1).

The *edit mask* operand (operand 3) is limited to no more than 256 bytes.

Mask Syntax: The source field is converted to packed decimal format. The *edit mask* contains both control character and data character strings. Both the *edit mask* and the source fields are processed left to right, and the edited result is placed in the result field from left to right. If the number of digits in the source field is even, the four high-order bits of the source field are ignored and not checked for validity. All other source digits as well as the sign are checked for validity, and a *decimal data* (hex 0C02) exception is signaled when one is invalid. Overlapping of any of these fields gives unpredictable results.

Nine fixed value control characters can be in the *edit mask*, hex AA through hex AD and hex AF through hex B3. Four of these control characters specify strings of characters to be inserted into the result field under certain conditions; and the other five indicate that a digit from the source field should be checked and the appropriate action taken.

One variable value control character can be in the *edit mask*. This control character indicates the end of a string of characters. The value of the end-of-string character can vary with each execution of the instruction and is determined by the value of the first character in the edit mask. If the first character of the *edit mask* is a value less than hex 40, then that value is used as the end-of-string character. If the first character of the *edit mask* is a value equal to or greater than hex 40, then hex AE is used as the end-of-string character.

A significance indicator is set to the off state at the start of the execution of this instruction. It remains in this state until a nonzero source digit is encountered in the source field or until one of the four unconditional digits (hex AA through hex AD) or an unconditional string (hex B3) is encountered in the *edit mask*.

When significance is detected, the selected floating string is overlaid into the result field immediately before (to the left of) the first significant result character.

When the significance indicator is set to the on state, the first significant result character has been reached. The state of the significance indicator determines whether the fill character or a digit from the source field is to be inserted into the result field for conditional digits and characters in conditional strings specified in the *edit mask* field. The fill character is a hex 40 until it is replaced by the first character following the floating string specification control character (hex B1).

When the significance indicator is in the off state:

-
- A conditional digit control character in the *edit mask* causes the fill character to be moved to the result field.
- A character in a conditional string in the *edit mask* causes the fill character to be moved to the result field.

When the significance indicator is in the on state:

-
- A conditional digit control character in the *edit mask* causes a source digit to be moved to the result field.
- A character in a conditional string in the *edit mask* is moved to the result field.

The following control characters are found in the *edit mask* field.

End-of-String Character: One of these control characters (a value less than hex 40 or hex AE) indicates the end of a character string and must be present even if the string is null.

Static Field Character:

Hex AF

This control character indicates the start of a static field. A static field is used to indicate that one of two mask character strings immediately following this character is to be inserted into the result field, depending upon the algebraic sign of the source field. If the sign is positive, the first string is to be inserted into the result field; if the sign is negative, the second string is to be inserted.

Static field format:

<Hex AF> <positive string>. . .<less than hex 40> <negative string>. . .<hex AE>

OR

<Hex AF> <positive string>. . .<hex AE> <negative string>. . .<hex AE>

Floating String Specification Field Character:

Hex B1

This control character indicates the start of a floating string specification field. The first character of the field is used as the fill character; following the fill character are two strings delimited by the end-of-string control character. If the algebraic sign of the source field is positive, the first string is to be overlaid into the result field; if the sign is negative, the second string is to be overlaid.

The string selected to be overlaid into the result field, called a floating string, appears immediately to the left of the first significant result character. If significance is never set, neither string is placed in the result field.

Conditional source digit positions (hex B2 control characters) must be provided in the *edit mask* immediately following the hex B1 field to accommodate the longer of the two floating strings; otherwise, a *length conformance* (hex 0C08) exception is signaled. For each of these B2 strings, the fill character is inserted into the result field, and source digits are not consumed. This ensures that the floating string never overlays bytes preceding the *receiver* operand.

Floating string specification field format:

<Hex B1> <fill character> <positive string>. . . <end-of-string character> <negative string>. . .<end-of-string character>

followed by

<Hex B2>. . .

Conditional String Character:

Hex B0

This control character indicates the start of a conditional string, which consists of any characters delimited by the end-of-string control character. Depending on the state of the significance indicator, this string or fill characters replacing it is inserted into the result field. If the significance indicator is off, a fill character for every character in the conditional string is placed in the result field. If the indicator is on, the characters in the conditional string are placed in the result field.

Conditional string format:

<Hex B0> <conditional string>. . .<end-of-string character>

Unconditional String Character:

Hex B3

This control character turns on the significance indicator and indicates the start of an unconditional string that consists of any characters delimited by the end-of-string control character. This string is unconditionally inserted into the result field regardless of the state of the significance indicator. If the indicator is off when a B3 control character is encountered, the appropriate floating string is overlaid into the result field before (to the left of) the B3 unconditional string (or to the left of where the unconditional string would have been if it were not null).

Unconditional string format:

<Hex B3> <unconditional string>. . .<end-of-string character>

Control Characters That Correspond to Digits in the Source Field:

Hex B2

This control character specifies that either the corresponding source field digit or the floating string (hex B1) fill character is inserted into the result field, depending on the state of the significance indicator. If the significance indicator is off, the fill character is placed in the result field; if the indicator is on, the source digit is placed. When a source digit is moved to the result field, the zone supplied is hex F. When significance (that is, a nonzero source digit) is detected, the floating string is overlaid to the left of the first significant character.

Control characters hex AA, hex AB, hex AC, and hex AD turn on the significance indicator. If the indicator is off when one of these control characters is encountered, the appropriate floating string is overlaid into the result field before (to the left of) the result digit.

Hex AA

This control character specifies that the corresponding source field digit is unconditionally placed in the 4 low-order bits of the result field with the zone set to a hex F.

Hex AB

This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the sign of the source field is positive, the zoned portion of the digit is set to hex F (the preferred positive sign); if the sign is negative, the zone portion is set to hex D (the preferred negative sign).

Hex AC

This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the algebraic sign of the source field is positive, the zone portion of the result is set to hex F (the preferred positive sign); otherwise, the source sign field is moved to the result zone field.

Hex AD

This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the algebraic sign of the source field is negative, the zone is set to hex D (the preferred negative sign); otherwise, the source field sign is moved to the zone position of the result byte.

The following table provides an overview of the results obtained with the valid edit conditions and sequences.

Table 1. Valid Edit Conditions and Results

Mask Character	Significance Indicator	Source Digit	Previous Significance Indicator	Result Character(s)	Resulting Significance Indicator
AF	Off	Any	Off	Positive string inserted	No Change
	On	Any	On	Negative string inserted	No Change
AA	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	On
	Off	0-9	Negative	Negative floating string overlaid; hex F, source digit	On
	On	0-9	Any	Hex F, source digit	On
AB	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	On
	Off	0-9	Negative	Negative floating string overlaid; hex D, source digit	On
	On	0-9	Positive	Hex F, source digit	On
	On	0-9	Negative	Hex D, source digit	On
AC	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	
	Off	0-9	Negative	Negative floating string overlaid; source sign and digit	On
	On	0-9	Positive	Hex F, source digit	On
	On	0-9	Negative	Source sign and digit	On
AD	Off	0-9	Positive	Positive floating string overlaid; source sign and digit	On
	Off	0-9	Negative	Negative floating string overlaid; hex D, source digit	On
	On	0-9	Positive	Source sign and digit	On
	On	0-9	Negative	Hex D, source digit	On
B0	Off	Any	Any	Insert fill character for each B0 string character	Off
	On	Any	Any	Insert B0 character string	On
B1 (including necessary B2s)	Off	Any	Any	Insert the fill character for each B2 character that corresponds to a character in the longer of the two floating strings	No Change
B2 (not for a B1 field)	Off	0	Any	Insert fill character	Off
	Off	1-9	Positive	Overlay positive floating string and insert hex F, source digit	On
	Off	1-9	Negative	Overlay negative floating string and insert hex F, source digit	On
	On	0-9	Any	Hex F, source digit	
B3	Off	Any	Positive	Overlay positive floating string and insert B3 character string	On
	Off	Any	Negative	Overlay negative floating string and insert B3 character string	On
	On	Any	Any	Insert B3 character string	On

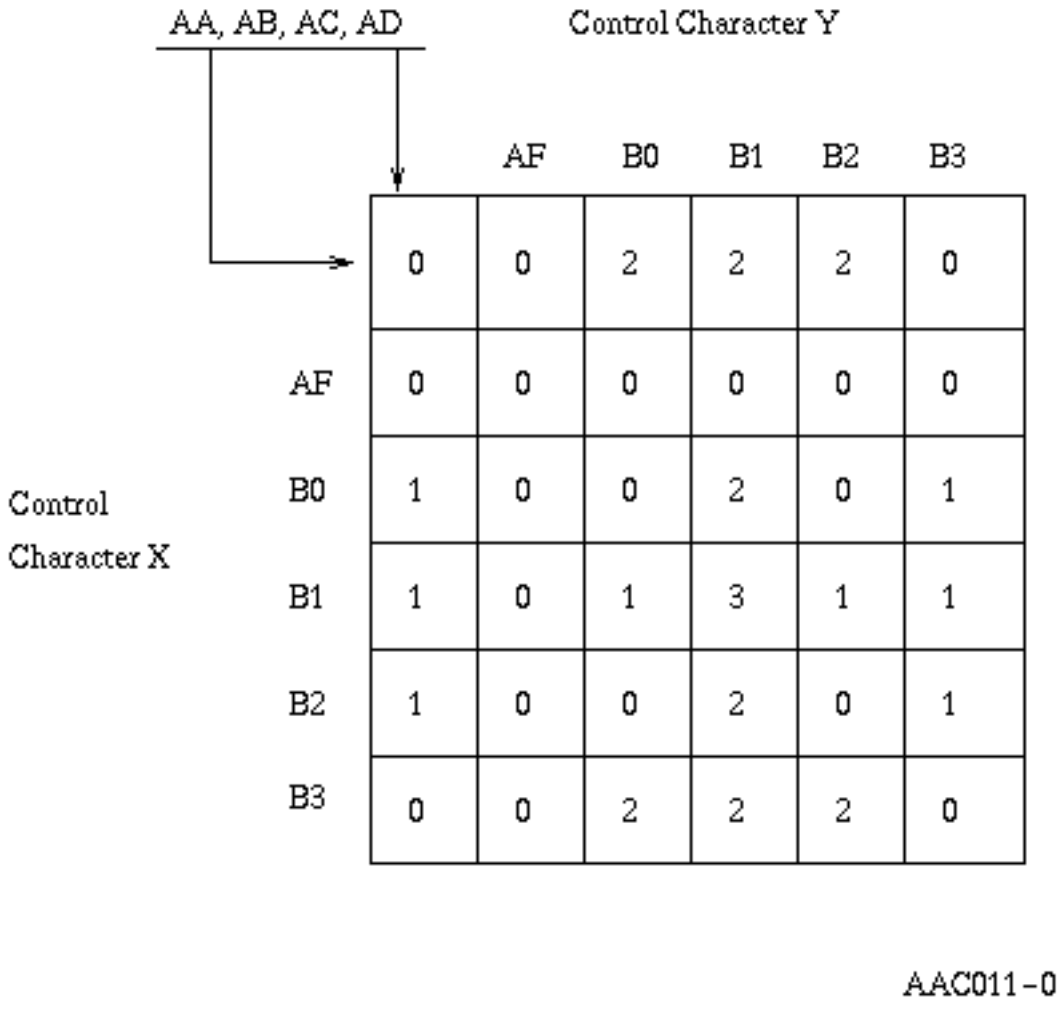
Mask Character	Previous Significance Indicator	Source Field Sign	Result Field Character(s)	Resulting Significance Indicator

Note:

1. Any character is a valid fill character, including the end-of-string character.
2. Hex AF, hex B1, hex B0, and hex B3 strings must be terminated by the end-of-string character even if they are null strings.
3. If a hex B1 field has not been encountered (specified) when the significance indicator is turned on, the floating string is considered to be a null string and is therefore not used to overlay into the result field.
4. If the positive and negative strings of a static field are of unequal length, additional static fields are necessary to ensure that the sum of the lengths of the positive strings equal the sum of the lengths of the negative strings; otherwise, a *length conformance* (hex 0C08) exception is signaled because the *receiver* length does not correspond to the length implied by the *edit mask* and source field sign.

The following figure indicates the valid ordering of control characters in an *edit mask* field.

Figure 1. Edit Mask Field Control Characters



Explanation:

Condition	Definition

0	Both X and Y can appear in the edit mask field in either order.
1	Y cannot precede X.
2	X cannot precede Y.
3	Both control characters (two B1's) cannot appear in an *edit mask* field.

Violation of any of the above rules will result in an *edit mask syntax* (hex 0C05) exception.

The following steps are performed when the editing is done:

•
- Convert Source Value to Packed Decimal
  –
  – The numeric value in the *source* operand is converted to a packed decimal intermediate value before the editing is done. If the *source* operand is binary, the attributes of the intermediate packed field before the edit are calculated as follows:

    Binary(2) = packed (5,0) or

    Binary(4) = packed (10,0)

    A data-pointer-defined *source* operand with 8 byte binary attributes is not supported and will cause a *scalar value invalid* (hex 3203) exception to be signaled.
- Edit
  –
  – The editing of the source digits and mask insertion characters into the *receiver* operand is done from left to right.
- Insert Floating String into Receiver Field
  –
  – If a floating string is to be inserted into the receiver field, this is done after the other editing.

Edit Digit Count Exception: An *edit digit count* (hex 0C04) exception is signaled when:

•
- The end of the source field is reached and there are more control characters that correspond to digits in the *edit mask* field.
- The end of the *edit mask* field is reached and there are more digit positions in the source field.

Edit Mask Syntax Exception: An *edit mask syntax* (hex 0C05) exception is signaled when an invalid *edit mask* control character is encountered or when a sequence rule is violated.

Length Conformance Exception: A *length conformance* (hex 0C08) exception is signaled when:

•
- The end of the *edit mask* field is reached and there are more character positions in the result field.
- The end of the result field is reached and more positions remain in the *edit mask* field.
- The number of B2s following a B1 field cannot accommodate the longer of the two floating strings.

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C04 Edit Digit Count

0C05 Edit Mask Syntax

0C08 Length Conformance

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Edit (EDIT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10E3	Receiver	Source	Edit mask

Operand 1: Character variable scalar or data-pointer-defined character scalar.

Operand 2: Numeric scalar or data-pointer-defined numeric scalar.

Operand 3: Character variable scalar or data-pointer-defined character scalar.

Bound program access

```
Built-in number for LBEDIT is 137.
LBEDIT (
        receiver            : address
        receiver_length     : address of unsigned binary(4)
        source              : address of signed binary(4) OR
                              address of unsigned binary(4) OR
                              address of packed decimal (1 to 63 digits) OR
                              address of zoned decimal (1 to 63 digits)
        source_attributes   : address
        mask                : address
        mask_length         : address of unsigned binary(4)
)
```

The *receiver*, *source* and *mask* parameters correspond to operands 1, 2 and 3 on the EDIT operation.

The *source_attributes* is a structure which describes the data attributes of the source value. The format of this structure matches that of the third operand on the CVTCN and CVTNC operations.

The *receiver_length* and *mask_length* parameters contain the length, in bytes, of the receiver and mask. They are expected to contain a value between 1 and 256.

-- OR --

```
EDITPD (
        receiver          : address
        receiver_length   : unsigned binary(4)
        source            : address of packed decimal (1 to 63 digits)
        source_length     : unsigned binary(4)
        mask              : address
        mask_length       : unsigned binary(4)
)
```

This built-in function supports the EDIT operation when the source contains a packed decimal value. The *receiver*, *source* and *mask* parameters correspond to operands 1, 2 and 3 on the EDIT operation.

The *source_length* parameter contains the length, in digits, of the source. It is expected to contain a value between 1 and 63.

The *receiver_length* and *mask_length* parameters contain the length, in bytes, of the receiver and mask. They are expected to contain a value between 1 and 256.

Description: The value of a numeric scalar is transformed from its internal form to character form suitable for display at a source/sink device. The following general editing functions can be performed during transforming of the *source* operand to the *receiver* operand:

-
- Unconditional insertion of a source value digit with a zone as a function of the source value's algebraic sign
- Unconditional insertion of a mask operand character string
- Conditional insertion of one of two possible *edit* mask operand character strings as a function of the source value's algebraic sign
- Conditional insertion of a source value digit or an *edit mask* operand replacement character as a function of source value leading zero suppression
- Conditional insertion of either an *edit mask* operand character string or a series of replacement characters as a function of source value leading zero suppression
- Conditional floating insertion of one of two possible *edit mask* operand character strings as a function of both the algebraic sign of the source value and leading zero suppression

The operation is performed by transforming the *source* (operand 2) under control of the *edit mask* (operand 3) and placing the result in the *receiver* (operand 1).

The *edit mask* operand (operand 3) is limited to no more than 256 bytes.

Mask Syntax: The source field is converted to packed decimal format. The *edit mask* contains both control character and data character strings. Both the *edit mask* and the source fields are processed left to right, and the edited result is placed in the result field from left to right. If the number of digits in the source field is even, the four high-order bits of the source field are ignored and not checked for validity. All other source digits as well as the sign are checked for validity, and a *decimal data* (hex 0C02) exception is signaled when one is invalid. Overlapping of any of these fields gives unpredictable results.

Nine fixed value control characters can be in the *edit mask*, hex AA through hex AD and hex AF through hex B3. Four of these control characters specify strings of characters to be inserted into the result field under certain conditions; and the other five indicate that a digit from the source field should be checked and the appropriate action taken.

One variable value control character can be in the *edit mask*. This control character indicates the end of a string of characters. The value of the end-of-string character can vary with each execution of the instruction and is determined by the value of the first character in the edit mask. If the first character of the *edit mask* is a value less than hex 40, then that value is used as the end-of-string character. If the first character of the *edit mask* is a value equal to or greater than hex 40, then hex AE is used as the end-of-string character.

A significance indicator is set to the off state at the start of the execution of this instruction. It remains in this state until a nonzero source digit is encountered in the source field or until one of the four unconditional digits (hex AA through hex AD) or an unconditional string (hex B3) is encountered in the *edit mask*.

When significance is detected, the selected floating string is overlaid into the result field immediately before (to the left of) the first significant result character.

When the significance indicator is set to the on state, the first significant result character has been reached. The state of the significance indicator determines whether the fill character or a digit from the source field is to be inserted into the result field for conditional digits and characters in conditional strings specified in the *edit mask* field. The fill character is a hex 40 until it is replaced by the first character following the floating string specification control character (hex B1).

When the significance indicator is in the off state:

-
- A conditional digit control character in the *edit mask* causes the fill character to be moved to the result field.
- A character in a conditional string in the *edit mask* causes the fill character to be moved to the result field.

When the significance indicator is in the on state:

-
- A conditional digit control character in the *edit mask* causes a source digit to be moved to the result field.
- A character in a conditional string in the *edit mask* is moved to the result field.

The following control characters are found in the *edit mask* field.

End-of-String Character: One of these control characters (a value less than hex 40 or hex AE) indicates the end of a character string and must be present even if the string is null.

Static Field Character:

Hex AF

This control character indicates the start of a static field. A static field is used to indicate that one of two mask character strings immediately following this character is to be inserted into the result field, depending upon the algebraic sign of the source field. If the sign is positive, the first string is to be inserted into the result field; if the sign is negative, the second string is to be inserted.

Static field format:

<Hex AF> <positive string>. . .<less than hex 40> <negative string>. . .<hex AE>

OR

<Hex AF> <positive string>. . .<hex AE> <negative string>. . .<hex AE>

Floating String Specification Field Character:

Hex B1

This control character indicates the start of a floating string specification field. The first character of the field is used as the fill character; following the fill character are two strings delimited by the end-of-string control character. If the algebraic sign of the source field is positive, the first string is to be overlaid into the result field; if the sign is negative, the second string is to be overlaid.

The string selected to be overlaid into the result field, called a floating string, appears immediately to the left of the first significant result character. If significance is never set, neither string is placed in the result field.

Conditional source digit positions (hex B2 control characters) must be provided in the *edit mask* immediately following the hex B1 field to accommodate the longer of the two floating strings; otherwise, a *length conformance* (hex 0C08) exception is signaled. For each of these B2 strings, the fill character is inserted into the result field, and source digits are not consumed. This ensures that the floating string never overlays bytes preceding the *receiver* operand.

Floating string specification field format:

<Hex B1> <fill character> <positive string>. . . <end-of-string character> <negative string>. . .<end-of-string character>

followed by

<Hex B2>. . .

Conditional String Character:

Hex B0

This control character indicates the start of a conditional string, which consists of any characters delimited by the end-of-string control character. Depending on the state of the significance indicator, this string or fill characters replacing it is inserted into the result field. If the significance indicator is off, a fill character for every character in the conditional string is placed in the result field. If the indicator is on, the characters in the conditional string are placed in the result field.

Conditional string format:

<Hex B0> <conditional string>. . .<end-of-string character>

Unconditional String Character:

Hex B3

This control character turns on the significance indicator and indicates the start of an unconditional string that consists of any characters delimited by the end-of-string control character. This string is unconditionally inserted into the result field regardless of the state of the significance indicator. If the indicator is off when a B3 control character is encountered, the appropriate floating string is overlaid into the result field before (to the left of) the B3 unconditional string (or to the left of where the unconditional string would have been if it were not null).

Unconditional string format:

<Hex B3> <unconditional string>. . .<end-of-string character>

Control Characters That Correspond to Digits in the Source Field:

Hex B2

This control character specifies that either the corresponding source field digit or the floating string (hex B1) fill character is inserted into the result field, depending on the state of the significance indicator. If the significance indicator is off, the fill character is placed in the result field; if the indicator is on, the source digit is placed. When a source digit is moved to the result field, the zone supplied is hex F. When significance (that is, a nonzero source digit) is detected, the floating string is overlaid to the left of the first significant character.

Control characters hex AA, hex AB, hex AC, and hex AD turn on the significance indicator. If the indicator is off when one of these control characters is encountered, the appropriate floating string is overlaid into the result field before (to the left of) the result digit.

Hex AA

This control character specifies that the corresponding source field digit is unconditionally placed in the 4 low-order bits of the result field with the zone set to a hex F.

Hex AB

This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the sign of the source field is positive, the zoned portion of the digit is set to hex F (the preferred positive sign); if the sign is negative, the zone portion is set to hex D (the preferred negative sign).

Hex AC

This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the algebraic sign of the source field is positive, the zone portion of the result is set to hex F (the preferred positive sign); otherwise, the source sign field is moved to the result zone field.

Hex AD

This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the algebraic sign of the source field is negative, the zone is set to hex D (the preferred negative sign); otherwise, the source field sign is moved to the zone position of the result byte.

The following table provides an overview of the results obtained with the valid edit conditions and sequences.

Table 1. Valid Edit Conditions and Results

Mask Character	Previous Significance Indicator	Source Digit	Source Sign	Result Character(s)	Resulting Significance Indicator
AF	Off	Any	Positive	Positive string inserted	No Change
	On	Any	Negative	Negative string inserted	No Change
AA	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	On
	Off	0-9	Negative	Negative floating string overlaid; hex F, source digit	On
	On	0-9	Any	Hex F, source digit	On
AB	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	On
	Off	0-9	Negative	Negative floating string overlaid; hex D, source digit	On
	On	0-9	Positive	Hex F, source digit	On
	On	0-9	Negative	Hex D, source digit	On
AC	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	
	Off	0-9	Negative	Negative floating string overlaid; source sign and digit	On
	On	0-9	Positive	Hex F, source digit	On
	On	0-9	Negative	Source sign and digit	On
AD	Off	0-9	Positive	Positive floating string overlaid; source sign and digit	On
	Off	0-9	Negative	Negative floating string overlaid; hex D, source digit	On
	On	0-9	Positive	Source sign and digit	On
	On	0-9	Negative	Hex D, source digit	On
B0	Off	Any	Any	Insert fill character for each B0 string character	Off
	On	Any	Any	Insert B0 character string	On
B1 (including necessary B2s)	Off	Any	Any	Insert the fill character for each B2 character that corresponds to a character in the longer of the two floating strings	No Change
B2 (not for a B1 field)	Off	0	Any	Insert fill character	Off
	Off	1-9	Positive	Overlay positive floating string and insert hex F, source digit	On
	Off	1-9	Negative	Overlay negative floating string and insert hex F, source digit	On
	On	0-9	Any	Hex F, source digit	
B3	Off	Any	Positive	Overlay positive floating string and insert B3 character string	On
	Off	Any	Negative	Overlay negative floating string and insert B3 character string	On
	On	Any	Any	Insert B3 character string	On

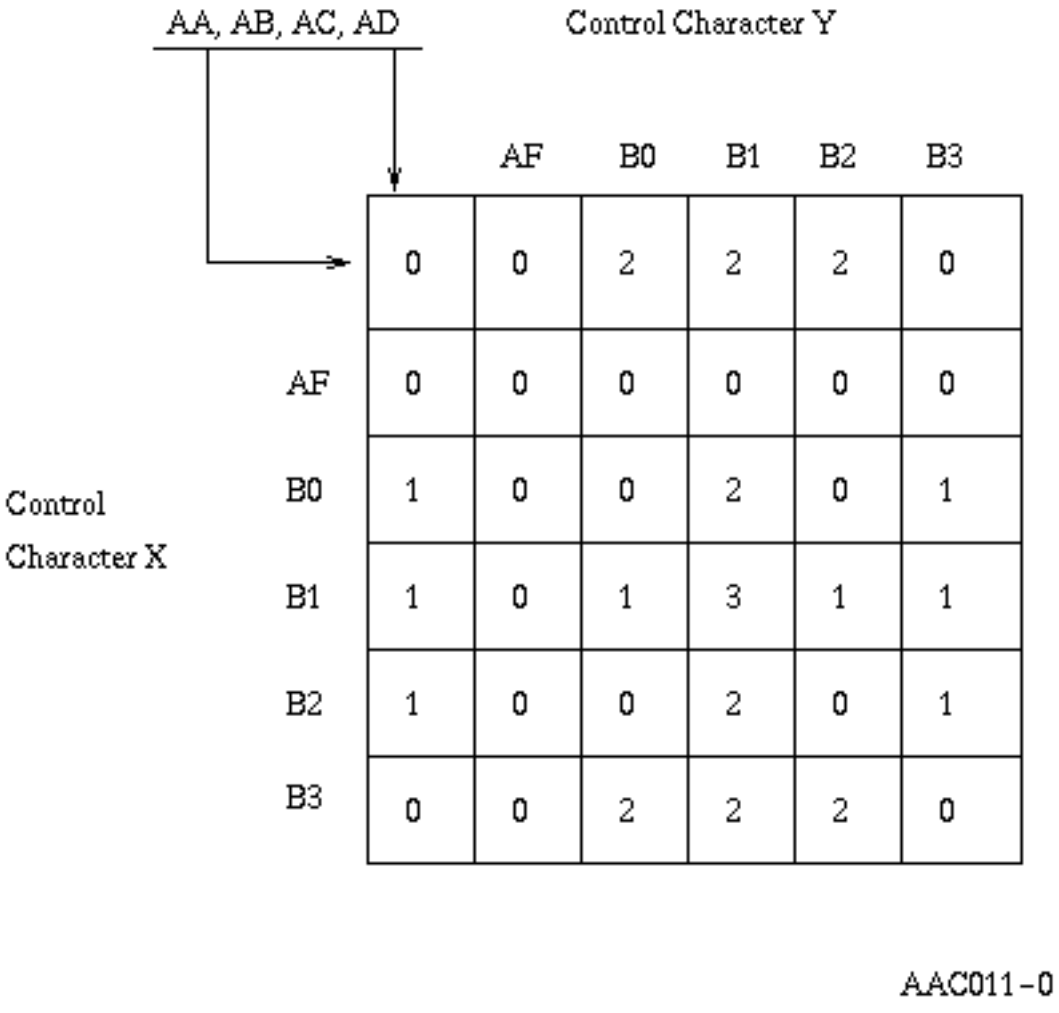
Mask Character(s)	Previous Significance Indicator	Source Field	Result Field	Resulting Significance Indicator

Note:

1. Any character is a valid fill character, including the end-of-string character.
2. Hex AF, hex B1, hex B0, and hex B3 strings must be terminated by the end-of-string character even if they are null strings.
3. If a hex B1 field has not been encountered (specified) when the significance indicator is turned on, the floating string is considered to be a null string and is therefore not used to overlay into the result field.
4. If the positive and negative strings of a static field are of unequal length, additional static fields are necessary to ensure that the sum of the lengths of the positive strings equal the sum of the lengths of the negative strings; otherwise, a *length conformance* (hex 0C08) exception is signaled because the *receiver* length does not correspond to the length implied by the *edit mask* and source field sign.

The following figure indicates the valid ordering of control characters in an *edit mask* field.

Figure 1. Edit Mask Field Control Characters



Explanation:

Condition	Definition

0	Both X and Y can appear in the edit mask field in either order.
1	Y cannot precede X.
2	X cannot precede Y.
3	Both control characters (two B1's) cannot appear in an *edit mask* field.

Violation of any of the above rules will result in an *edit mask syntax* (hex 0C05) exception.

The following steps are performed when the editing is done:

•
- Convert Source Value to Packed Decimal
  –
  – The numeric value in the *source* operand is converted to a packed decimal intermediate value before the editing is done. If the *source* operand is binary, the attributes of the intermediate packed field before the edit are calculated as follows:

    Binary(2) = packed (5,0) or

    Binary(4) = packed (10,0)

    A data-pointer-defined *source* operand with 8 byte binary attributes is not supported and will cause a *scalar value invalid* (hex 3203) exception to be signaled.
- Edit
  –
  – The editing of the source digits and mask insertion characters into the *receiver* operand is done from left to right.
- Insert Floating String into Receiver Field
  –
  – If a floating string is to be inserted into the receiver field, this is done after the other editing.

Edit Digit Count Exception: An *edit digit count* (hex 0C04) exception is signaled when:

•
- The end of the source field is reached and there are more control characters that correspond to digits in the *edit mask* field.
- The end of the *edit mask* field is reached and there are more digit positions in the source field.

Edit Mask Syntax Exception: An *edit mask syntax* (hex 0C05) exception is signaled when an invalid *edit mask* control character is encountered or when a sequence rule is violated.

Length Conformance Exception: A *length conformance* (hex 0C08) exception is signaled when:

•
- The end of the *edit mask* field is reached and there are more character positions in the result field.
- The end of the result field is reached and more positions remain in the *edit mask* field.
- The number of B2s following a B1 field cannot accommodate the longer of the two floating strings.

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C04 Edit Digit Count

0C05 Edit Mask Syntax

0C08 Length Conformance

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Edit (EDIT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10E3	Receiver	Source	Edit mask

Operand 1: Character variable scalar or data-pointer-defined character scalar.

Operand 2: Numeric scalar or data-pointer-defined numeric scalar.

Operand 3: Character variable scalar or data-pointer-defined character scalar.

Bound program access

```
Built-in number for LBEDIT is 137.
LBEDIT (
        receiver            : address
        receiver_length     : address of unsigned binary(4)
        source              : address of signed binary(4) OR
                              address of unsigned binary(4) OR
                              address of packed decimal (1 to 63 digits) OR
                              address of zoned decimal (1 to 63 digits)
        source_attributes   : address
        mask                : address
        mask_length         : address of unsigned binary(4)
)
```

The *receiver*, *source* and *mask* parameters correspond to operands 1, 2 and 3 on the EDIT operation.

The *source_attributes* is a structure which describes the data attributes of the source value. The format of this structure matches that of the third operand on the CVTCN and CVTNC operations.

The *receiver_length* and *mask_length* parameters contain the length, in bytes, of the receiver and mask. They are expected to contain a value between 1 and 256.

-- OR --

```
EDITPD (
        receiver          : address
        receiver_length   : unsigned binary(4)
        source            : address of packed decimal (1 to 63 digits)
        source_length     : unsigned binary(4)
        mask              : address
        mask_length       : unsigned binary(4)
)
```

This built-in function supports the EDIT operation when the source contains a packed decimal value. The *receiver*, *source* and *mask* parameters correspond to operands 1, 2 and 3 on the EDIT operation.

The *source_length* parameter contains the length, in digits, of the source. It is expected to contain a value between 1 and 63.

The *receiver_length* and *mask_length* parameters contain the length, in bytes, of the receiver and mask. They are expected to contain a value between 1 and 256.

Description: The value of a numeric scalar is transformed from its internal form to character form suitable for display at a source/sink device. The following general editing functions can be performed during transforming of the *source* operand to the *receiver* operand:

-
- Unconditional insertion of a source value digit with a zone as a function of the source value's algebraic sign
- Unconditional insertion of a mask operand character string
- Conditional insertion of one of two possible *edit* mask operand character strings as a function of the source value's algebraic sign
- Conditional insertion of a source value digit or an *edit mask* operand replacement character as a function of source value leading zero suppression
- Conditional insertion of either an *edit mask* operand character string or a series of replacement characters as a function of source value leading zero suppression
- Conditional floating insertion of one of two possible *edit mask* operand character strings as a function of both the algebraic sign of the source value and leading zero suppression

The operation is performed by transforming the *source* (operand 2) under control of the *edit mask* (operand 3) and placing the result in the *receiver* (operand 1).

The *edit mask* operand (operand 3) is limited to no more than 256 bytes.

Mask Syntax: The source field is converted to packed decimal format. The *edit mask* contains both control character and data character strings. Both the *edit mask* and the source fields are processed left to right, and the edited result is placed in the result field from left to right. If the number of digits in the source field is even, the four high-order bits of the source field are ignored and not checked for validity. All other source digits as well as the sign are checked for validity, and a *decimal data* (hex 0C02) exception is signaled when one is invalid. Overlapping of any of these fields gives unpredictable results.

Nine fixed value control characters can be in the *edit mask*, hex AA through hex AD and hex AF through hex B3. Four of these control characters specify strings of characters to be inserted into the result field under certain conditions; and the other five indicate that a digit from the source field should be checked and the appropriate action taken.

One variable value control character can be in the *edit mask*. This control character indicates the end of a string of characters. The value of the end-of-string character can vary with each execution of the instruction and is determined by the value of the first character in the edit mask. If the first character of the *edit mask* is a value less than hex 40, then that value is used as the end-of-string character. If the first character of the *edit mask* is a value equal to or greater than hex 40, then hex AE is used as the end-of-string character.

A significance indicator is set to the off state at the start of the execution of this instruction. It remains in this state until a nonzero source digit is encountered in the source field or until one of the four unconditional digits (hex AA through hex AD) or an unconditional string (hex B3) is encountered in the *edit mask*.

When significance is detected, the selected floating string is overlaid into the result field immediately before (to the left of) the first significant result character.

When the significance indicator is set to the on state, the first significant result character has been reached. The state of the significance indicator determines whether the fill character or a digit from the source field is to be inserted into the result field for conditional digits and characters in conditional strings specified in the *edit mask* field. The fill character is a hex 40 until it is replaced by the first character following the floating string specification control character (hex B1).

When the significance indicator is in the off state:

-
- A conditional digit control character in the *edit mask* causes the fill character to be moved to the result field.
- A character in a conditional string in the *edit mask* causes the fill character to be moved to the result field.

When the significance indicator is in the on state:

-
- A conditional digit control character in the *edit mask* causes a source digit to be moved to the result field.
- A character in a conditional string in the *edit mask* is moved to the result field.

The following control characters are found in the *edit mask* field.

End-of-String Character: One of these control characters (a value less than hex 40 or hex AE) indicates the end of a character string and must be present even if the string is null.

Static Field Character:

Hex AF

This control character indicates the start of a static field. A static field is used to indicate that one of two mask character strings immediately following this character is to be inserted into the result field, depending upon the algebraic sign of the source field. If the sign is positive, the first string is to be inserted into the result field; if the sign is negative, the second string is to be inserted.

Static field format:

<Hex AF> <positive string>. . .<less than hex 40> <negative string>. . .<hex AE>

OR

<Hex AF> <positive string>. . .<hex AE> <negative string>. . .<hex AE>

Floating String Specification Field Character:

Hex B1

This control character indicates the start of a floating string specification field. The first character of the field is used as the fill character; following the fill character are two strings delimited by the end-of-string control character. If the algebraic sign of the source field is positive, the first string is to be overlaid into the result field; if the sign is negative, the second string is to be overlaid.

The string selected to be overlaid into the result field, called a floating string, appears immediately to the left of the first significant result character. If significance is never set, neither string is placed in the result field.

Conditional source digit positions (hex B2 control characters) must be provided in the *edit mask* immediately following the hex B1 field to accommodate the longer of the two floating strings; otherwise, a *length conformance* (hex 0C08) exception is signaled. For each of these B2 strings, the fill character is inserted into the result field, and source digits are not consumed. This ensures that the floating string never overlays bytes preceding the *receiver* operand.

Floating string specification field format:

<Hex B1> <fill character> <positive string>. . . <end-of-string character> <negative string>. . .<end-of-string character>

followed by

<Hex B2>. . .

Conditional String Character:

Hex B0

This control character indicates the start of a conditional string, which consists of any characters delimited by the end-of-string control character. Depending on the state of the significance indicator, this string or fill characters replacing it is inserted into the result field. If the significance indicator is off, a fill character for every character in the conditional string is placed in the result field. If the indicator is on, the characters in the conditional string are placed in the result field.

Conditional string format:

<Hex B0> <conditional string>. . .<end-of-string character>

Unconditional String Character:

Hex B3

This control character turns on the significance indicator and indicates the start of an unconditional string that consists of any characters delimited by the end-of-string control character. This string is unconditionally inserted into the result field regardless of the state of the significance indicator. If the indicator is off when a B3 control character is encountered, the appropriate floating string is overlaid into the result field before (to the left of) the B3 unconditional string (or to the left of where the unconditional string would have been if it were not null).

Unconditional string format:

<Hex B3> <unconditional string>. . .<end-of-string character>

Control Characters That Correspond to Digits in the Source Field:

Hex B2

This control character specifies that either the corresponding source field digit or the floating string (hex B1) fill character is inserted into the result field, depending on the state of the significance indicator. If the significance indicator is off, the fill character is placed in the result field; if the indicator is on, the source digit is placed. When a source digit is moved to the result field, the zone supplied is hex F. When significance (that is, a nonzero source digit) is detected, the floating string is overlaid to the left of the first significant character.

Control characters hex AA, hex AB, hex AC, and hex AD turn on the significance indicator. If the indicator is off when one of these control characters is encountered, the appropriate floating string is overlaid into the result field before (to the left of) the result digit.

Hex AA

This control character specifies that the corresponding source field digit is unconditionally placed in the 4 low-order bits of the result field with the zone set to a hex F.

Hex AB

This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the sign of the source field is positive, the zoned portion of the digit is set to hex F (the preferred positive sign); if the sign is negative, the zone portion is set to hex D (the preferred negative sign).

Hex AC

This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the algebraic sign of the source field is positive, the zone portion of the result is set to hex F (the preferred positive sign); otherwise, the source sign field is moved to the result zone field.

Hex AD

This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the algebraic sign of the source field is negative, the zone is set to hex D (the preferred negative sign); otherwise, the source field sign is moved to the zone position of the result byte.

The following table provides an overview of the results obtained with the valid edit conditions and sequences.

Table 1. Valid Edit Conditions and Results

Mask Character	Significance Indicator	Previous Source Digit	Source Sign	Result Sign	Resulting Significance Indicator
AF	Off	Any	Positive	Positive string inserted	No Change
			Negative	Negative string inserted	No Change
AA	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	On
			Negative	Negative floating string overlaid; hex F, source digit	On
	On	0-9	Any	Hex F, source digit	On
AB	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	On
			Negative	Negative floating string overlaid; hex D, source digit	On
	On	0-9	Positive	Hex F, source digit	On
			Negative	Hex D, source digit	On
AC	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	
			Negative	Negative floating string overlaid; source sign and digit	On
	On	0-9	Positive	Hex F, source digit	On
			Negative	Source sign and digit	On
AD	Off	0-9	Positive	Positive floating string overlaid; source sign and digit	On
			Negative	Negative floating string overlaid; hex D, source digit	On
	On	0-9	Positive	Source sign and digit	On
			Negative	Hex D, source digit	On
B0	Off	Any	Any	Insert fill character for each B0 string character	Off
	On	Any	Any	Insert B0 character string	On
B1 (including necessary B2s)	Off	Any	Any	Insert the fill character for each B2 character that corresponds to a character in the longer of the two floating strings	No Change
B2 (not for a B1 field)	Off	0	Any	Insert fill character	Off
	Off	1-9	Positive	Overlay positive floating string and insert hex F, source digit	On
	Off	1-9	Negative	Overlay negative floating string and insert hex F, source digit	On
	On	0-9	Any	Hex F, source digit	
B3	Off	Any	Positive	Overlay positive floating string and insert B3 character string	On
	Off	Any	Negative	Overlay negative floating string and insert B3 character string	On
	On	Any	Any	Insert B3 character string	On

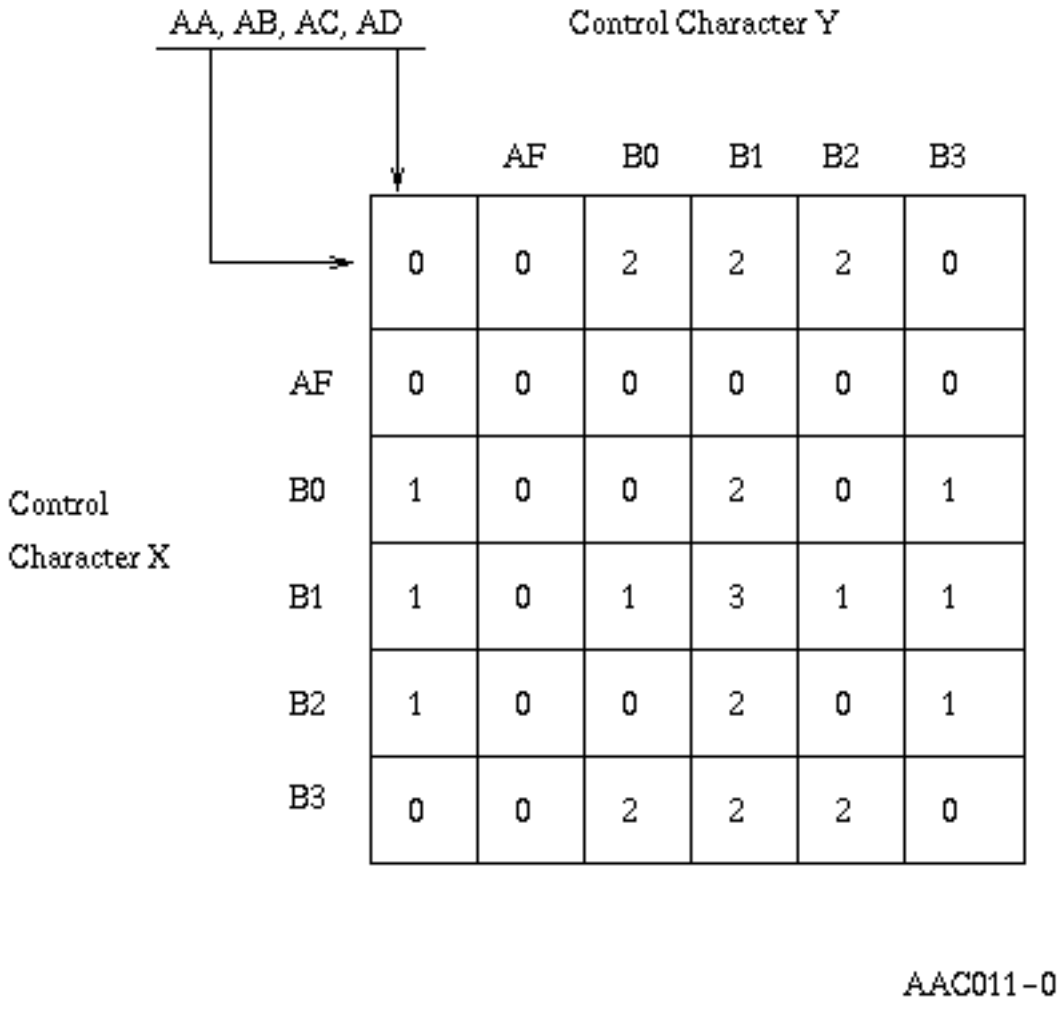
Mask Character	Previous Significance Indicator	Source Field Sign	Result Field Character(s)	Resulting Significance Indicator

Note:

1. Any character is a valid fill character, including the end-of-string character.
2. Hex AF, hex B1, hex B0, and hex B3 strings must be terminated by the end-of-string character even if they are null strings.
3. If a hex B1 field has not been encountered (specified) when the significance indicator is turned on, the floating string is considered to be a null string and is therefore not used to overlay into the result field.
4. If the positive and negative strings of a static field are of unequal length, additional static fields are necessary to ensure that the sum of the lengths of the positive strings equal the sum of the lengths of the negative strings; otherwise, a *length conformance* (hex 0C08) exception is signaled because the *receiver* length does not correspond to the length implied by the *edit mask* and source field sign.

The following figure indicates the valid ordering of control characters in an *edit mask* field.

Figure 1. Edit Mask Field Control Characters



Explanation:

Condition	Definition

0	Both X and Y can appear in the edit mask field in either order.
1	Y cannot precede X.
2	X cannot precede Y.
3	Both control characters (two B1's) cannot appear in an *edit mask* field.

Violation of any of the above rules will result in an *edit mask syntax* (hex 0C05) exception.

The following steps are performed when the editing is done:

•
- Convert Source Value to Packed Decimal
  –
  – The numeric value in the *source* operand is converted to a packed decimal intermediate value before the editing is done. If the *source* operand is binary, the attributes of the intermediate packed field before the edit are calculated as follows:

    Binary(2) = packed (5,0) or

    Binary(4) = packed (10,0)

    A data-pointer-defined *source* operand with 8 byte binary attributes is not supported and will cause a *scalar value invalid* (hex 3203) exception to be signaled.
- Edit
  –
  – The editing of the source digits and mask insertion characters into the *receiver* operand is done from left to right.
- Insert Floating String into Receiver Field
  –
  – If a floating string is to be inserted into the receiver field, this is done after the other editing.

Edit Digit Count Exception: An *edit digit count* (hex 0C04) exception is signaled when:

•
- The end of the source field is reached and there are more control characters that correspond to digits in the *edit mask* field.
- The end of the *edit mask* field is reached and there are more digit positions in the source field.

Edit Mask Syntax Exception: An *edit mask syntax* (hex 0C05) exception is signaled when an invalid *edit mask* control character is encountered or when a sequence rule is violated.

Length Conformance Exception: A *length conformance* (hex 0C08) exception is signaled when:

•
- The end of the *edit mask* field is reached and there are more character positions in the result field.
- The end of the result field is reached and more positions remain in the *edit mask* field.
- The number of B2s following a B1 field cannot accommodate the longer of the two floating strings.

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C04 Edit Digit Count

0C05 Edit Mask Syntax

0C08 Length Conformance

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

End (END)

Op Code (Hex)
0260

Description: The instruction delimits the end of a program's instruction stream. When this instruction is encountered in execution, it causes a return to the preceding invocation (if present) or causes termination of the process phase if the instruction is executed in the highest-level invocation for the initial thread of the process. The End instruction delineates the end of the instruction stream. When it is encountered in execution, the instruction functions as a Return External instruction with a null operand. Refer to the Return External (RTX) instruction for a description of that instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2208 Object Compressed

220B Object Not Available

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

Enqueue (ENQ)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
036B	Queue	Message prefix	Message text

Operand 1: System pointer.

Operand 2: Character scalar.

Operand 3: Space pointer.

Bound program access

```
Built-in number for ENQ is 43.
ENQ (
     queue           : address of system pointer
     message_prefix  : address
     message_text    : address
)
```

Description: A message is enqueued according to the *queue type* attribute specified during the queue's creation.

If *keyed* sequence is specified, enqueued messages are sequenced in ascending binary collating order according to the key value. If a message to be enqueued has a key value equal to an existing enqueued key value, the message being added is enqueued following the existing message.

If the queue was defined with either *last in, first out (LIFO)* or *first in, first out (FIFO)* sequencing, then enqueued messages are ordered chronologically with the latest enqueued message being either first on the queue or last on the queue, respectively. A key can be provided and associated with messages enqueued in a LIFO or FIFO queue; however, the key does not establish a message's position in the queue. The key can contain pointers, but the pointers are not considered to be pointers when they are placed on the queue by an Enqueue instruction.

Operand 1 specifies the *queue* to which a message is to be enqueued. Operand 2 specifies the *message prefix*, and operand 3 specifies the *message text*.

The format of the *message prefix* is as follows:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Size of message to be enqueued	Bin(4)
4	4	Enqueue key value (ignored for FIFO/LIFO queues with key lengths equal to 0)	Char(key length)
*	*	—- End —-	

The **size of message to be enqueued** is supplied to inform the machine of the number of bytes in the space that are to be considered *message text*. The size of the message is then considered the lesser of the *size of message to be enqueued* attribute and the *maximum message size* specified on queue creation. The *message text* can contain pointers. When pointers are in *message text*, the operand 3 space pointer must be 16-byte aligned. Improper alignment will result in an exception being signaled.

If the enqueued message causes the number of messages to exceed the maximum number of messages attribute of the queue, one of the following occurs:

•
- If the queue is not extendable, or if the maximum number of extends specified by the user at queue creation has been reached, then the *queue full* (hex 2602) exception is signaled. The message is not enqueued.
- If the queue is extendable, the queue is implicitly extended by the *extension value* attribute. The message is enqueued. No exception is signaled.

Limitations (Subject to Change): The maximum allowable queue size, including all messages currently enqueued and the machine overhead, is 2 gigabytes.

Warning: Temporary Level 3 Header

Authorization Required

•
- Insert
  –
  – Operand 1
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize

  –

  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C0E IASP Resources Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

26 Process Management

2602 Queue full

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

30 Journal

3002 Entry Not Journaled

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Ensure Object (ENSOBJ)

Op Code (Hex)	Operand 1
0381	System object

Operand 1: System pointer.

Bound program access
Built-in number for ENSOBJ is 67. ENSOBJ (system_object : address of system pointer)

Description: The *system object* identified by operand 1 is protected from volatile storage loss. The machine ensures that any changes made to the specified object are recorded on nonvolatile storage media. The access state of the object is not changed by this instruction. If operand 1 addresses a temporary

object, no operation is performed because temporary objects are not preserved during a machine failure. No exception is signaled if temporary objects are referenced.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2204 Object Not Eligible for Operation

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

30 Journal

3002 Entry Not Journaled

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Exchange Bytes (EXCHBY)

Op Code (Hex)	Operand 1	Operand 2
10CE	Source 1	Source 2

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character variable scalar or numeric variable scalar.

Description: The logical character string values of the two source operands are exchanged. The value of the second source operand is placed in the first source operand and the value of the first source operand is placed in the second operand.

The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings. Both operands must have the same length.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Exclusive Or (XOR)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
XOR 109B		Receiver	Source 1	Source 2	
XORI 189B	Indicator options	Receiver	Source 1	Source 2	Indicator targets
XORB 1C9B	Branch options	Receiver	Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character scalar or numeric scalar.

Operand 4-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms:

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-4]
XORS 119B		Receiver/Source 1	Source 2	
XORIS 199B	Indicator options	Receiver/Source 1	Source 2	Indicator targets

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-4]
XORBS 1D9B	Branch options	Receiver/Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3-4:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Boolean **exclusive or** operation is performed on the string values in the source operands. The resulting string is placed in the *receiver* operand.

The operands may be character or numeric scalars. They are both interpreted as bit strings. Substringing is supported for both character and numeric operands.

The length of the operation is equal to the length of the longer of the two source operands. The shorter of the two operands is padded on the right. The operation begins with the two source operands left-adjusted and continues bit by bit until they are completed.

The bit values of the result are determined as follows:

Source 1 Bit	Source 2 Bit	Result Bit
0	0	0
0	1	1
1	0	1
1	1	0

The result value is then placed (left-adjusted) in the *receiver* operand with truncating or padding taking place on the right.

The pad value used in this instruction is a hex 00.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1, 2, and 3. The effect of specifying a null substring reference for one source operand is that the other source operand is **exclusive or**ed with an equal length string of all hex 00s. When a null substring reference is specified for both source operands, the result is all zero and the instruction's resultant condition is zero. When a null substring reference is specified for the *receiver*, a result is not set and the instruction's resultant condition is zero regardless of the values of the source operands.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

When the *receiver* operand is a numeric variable scalar, it is possible that the result produced will not be a valid value for the numeric type. This can occur due to padding with hex 00, due to truncation, or due to the resultant bit string produced by the instruction. The instruction completes normally and signals no exceptions for these conditions.

Resultant Conditions:

•

- Zero-The bit value for the bits of the scalar *receiver* operand is either all zero or a null substring reference is specified for the *receiver*.
- Not zero-The bit value for the bits of the scalar *receiver* operand is not all zero.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Exponential Function of E (EEXP)

Bound program access
Built-in number for EEXP is 405. EEXP (source : floating point(8) value) : floating point(8) value which is e raised to the power of the source value

Description: The computation e ^{source} is performed and the result returned.

The result is in the range:

```
0 <= EEXP(source) <= +infinity
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Extended Character Scan (ECSCAN)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-7]
ECSCAN 10D4		Receiver	Base	Compare operand	Mode operand	
ECSCANB 1CD4	Branch options	Receiver	Base	Compare operand	Mode operand	Branch targets
ECSCANI 18D4	Indicator options	Receiver	Base	Compare operand	Mode operand	Indicator targets

Operand 1: Binary variable scalar or binary array.

Operand 2: Character variable scalar.

Operand 3: Character scalar.

Operand 4: Character(1) scalar.

Operand 5-7:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction scans the string value of the *base* operand for occurrences of the string value of the *compare* operand and indicates the relative locations of these occurrences in the *receiver* operand. The character string value of the *base* operand is scanned for occurrences of the character string value of the *compare* operand under control of the *mode* operand and mode control characters embedded in the base string.

The *base* and *compare* operands must both be character strings. The length of the *compare* operand must not be greater than that of the base string. The *base* and *compare* operand are interpreted as containing a mixture of 1-byte (simple) and 2-byte (extended) character codes. The mode, simple or extended, with which the string is to be interpreted, is controlled initially by the *mode* operand and thereafter by mode control characters embedded in the strings. The mode control characters are as follows:

* Hex 0E =	Shift out of simple character mode to extended mode.
* Hex 0F =	Shift into simple character mode from extended mode. This is recognized only if it occurs in the first byte position of an extended character code.

The format of the *mode* operand is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Mode operand	Char(1)		
0	0		Operand 2 initial mode indicator		Bit 0
			0 =	Operand starts in simple character mode.	
			1 =	Operand starts in extended character mode.	
0	0		Operand 3 initial mode indicator		Bit 1
			0 =	Operand starts in simple character mode.	
			1 =	Operand starts in extended character mode.	
0	0		Reserved (binary 0)		Bits 2-7
1	1	—- End —-			

The operation begins at the left end of the base string and continues character by character, left to right. When the base string is interpreted in *simple character mode*, the operation moves through the base string 1 byte at a time. When the base string is interpreted in *extended character mode*, the operation moves through the base string 2 bytes at a time.

The *compare* operand value is the entire byte string specified for the *compare* operand. The *mode* operand determines the initial mode of the *compare* operand. The first character of the *compare* operand value is assumed to be a valid character for the initial mode of the *compare* operand and not a mode control character. Mode control characters in the *compare* operand value participate in comparisons performed during the scan function except that a mode control character as the first character of the *compare* operand causes unpredictable results.

The base string is scanned until the mode of the characters being processed is the same as the initial mode of the *compare* operand value. The operation continues comparing the characters of the base string with those of the *compare* operand value. The starting character of the characters being compared in the base string is always a valid character for the initial mode of the *compare* operand value. A mode control character encountered in the base string that changed the base string mode to match the initial mode of the *compare* operand value does not participate in the comparison. The length of the comparison is equal to the length of the *compare* operand value and the comparison is performed the same as performed by the Compare Bytes Left Adjusted (CMPBLA) instruction.

If a set of bytes that matches the *compare* operand value is found, the binary value for the relative location of the leftmost base string character of the set of bytes is placed in the *receiver* operand.

If the *receiver* operand is a scalar, only the first occurrence of the *compare* operand is noted. If the *receiver* operand is an array, as many occurrences as there are elements in the array are noted.

If a mode change is encountered in the base string, the base string is again scanned until the mode of the characters being processed is the same as the initial mode of the *compare* operand value, and then the comparisons are resumed.

The operation continues until no more occurrences of the *compare* operand value can be noted in the *receiver* operand or until the number of bytes remaining to be scanned in the base string is less than the

length of the *compare* operand value. When the second condition occurs, the *receiver* value is set to zero. If the *receiver* operand is an array, all its remaining elements are also set to zero.

If the *escape code encountered* result condition is specified (through a branch or indicator option), verifications are performed on the base string as it is scanned. Each byte of the base string is checked for a value less than hex 40. When a value less than hex 40 is encountered, it is then determined if it is a valid mode control character.

If a byte value of less than hex 40 is not a valid mode control character, it is considered to be an escape code. The binary value for the relative location of the character (simple or extended) being interrogated is placed in the *receiver* operand, and the appropriate action (indicator or branch) is performed according to the specification of the escape code encountered result condition. If the *receiver* operand is an array, the next array element after any elements set with locations or prior occurrences of the *compare* operand, is set with the location of the character containing the escape code and all the remaining array elements are set to zero.

If the *escape encountered* result condition is not specified, verifications of the character codes in the base string are not performed.

Resultant Conditions:

•

- Positive-The numeric value(s) of the *receiver* operand is positive.
- Zero-The numeric value(s) of the *receiver* operand is zero. In the case where the *receiver* operand is an array, the resultant condition is zero if all elements are zero.
- Escape code encountered-An escape character code value was encountered during the scanning of the base string.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C08 Length Conformance

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2C Program Execution

- 2C04 Branch Target Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

- 3201 Scalar Type Invalid
- 3203 Scalar Value Invalid

36 Space Management

- 3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Extract Exponent (EXTREXP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
EXTREXP 1072		Receiver	Source	
EXTREXPB 1C72	Branch options	Receiver	Source	Branch targets
EXTREXPI 1872	Indicator options	Receiver	Source	Indicator targets

Operand 1: Binary variable scalar.

Operand 2: Floating-point scalar.

Operand 3-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction extracts the exponent portion of a floating-point scalar *source* operand and places it into the *receiver* operand as a binary variable scalar.

The operands must be the numeric types indicated because no conversions are performed.

The *source* floating-point field is interrogated to determine the binary floating-point value represented and either a signed exponent, for number values, or a special identifier, for infinity and NaN values, is placed in the binary variable scalar *receiver* operand.

The value to be assigned to the *receiver* is dependent upon the floating-point value represented in the *source* operand as described below. It is a signed binary integer value and a numeric assignment of the value is made to the *receiver*.

When the *source* represents a normalized number, the biased exponent contained in the exponent field of the *source* is converted to the corresponding signed exponent by subtracting the bias of 127 for short or 1,023 for long to determine the value to be returned. The resulting value ranges from -126 to +127 for short format, -1,022 to +1,023 for long format. When the *receiver* is unsigned binary, a negative exponent will result in a *size* (hex 0C0A) exception unless size exceptions are suppressed.

When the *source* represents a denormalized number, the value to be returned is determined by adjusting the signed exponent of the denormalized number. The signed exponent of a denormalized number is a fixed value of -126 for the short format and -1,022 for the long format. It is adjusted to the value the signed exponent would be if the *source* value was adjusted to a normalized number. The resulting value ranges from -127 to -149 for short format, -1,023 to -1,074 for long format.

When the *source* represents a value of zero, the value returned is zero.

When the *source* represents infinity, the value returned is +32,767.

When the *source* represents a not-a-number, the value returned is -32,768 for a signed binary receiver. For an unsigned binary(2) a value of 32,768 is returned, and for a unsigned binary(4) a value of 4,294,934,528 is returned.

Resultant Conditions:

•

- Normalized-The *source* operand value represents a normalized binary floating-point number. The signed exponent is stored in the *receiver*.
- Denormalized-The *source* operand value represents a denormalized binary floating-point number. An adjusted signed exponent is stored in the *receiver*.
- Infinity-The *source* operand value represents infinity. The *receiver* is set with a value of +32,767.
- NaN-The *source* operand value represents a not-a-number. The *receiver* is set with a value of -32,768 when signed binary, with a value of 32,768 when unsigned binary(2), and with a value of 4,294,934,528 when unsigned binary(4).

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

0C Computation

0C0A Size

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Extract Magnitude (EXTRMAG)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
EXTRMAG 1052		Receiver	Source	
EXTRMAGI 1852	Indicator options	Receiver	Source	Indicator targets
EXTRMAGB 1C52	Branch options	Receiver	Source	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand [2-5]
EXTRMAGS 1152		Receiver/Source	
EXTRMAGIS 1952	Indicator options	Receiver/Source	Indicator targets
EXTRMAGBS 1D52	Branch options	Receiver/Source	Branch targets

Operand 1: Numeric variable scalar.

Operand 2-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The numeric value of the *source* operand is converted to its absolute value and placed in the numeric variable scalar *receiver* operand.

The absolute value is formed from the *source* operand as follows:

•

- Signed binary
  –
  – Extract the numeric value and form twos complement if the source operand is negative.
- Unsigned signed binary
  –
  – Extract the numeric value.
- Packed/Zoned
  –
  – Extract the numeric value and force the *source* operand's sign to positive.
- Floating-point
  –
  – Extract the numeric value and force the significand sign to positive.

The result of the operation is copied into the *receiver* operand according to the rules of the Copy Numeric Value (CPYNV) instruction. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the *receiver* operand, or aligned at the assumed decimal point of the *receiver* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations outlined in Arithmetic Operations. If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled. An attempt to extract the magnitude of a maximum negative binary value to a binary scalar of the same size also results in a *size* (hex 0C0A) exception.

When the *source* floating-point operand represents not-a-number, the sign field of the *source* is not forced to positive and this value is not altered in the *receiver*.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

For a fixed-point operation, if significant digits are truncated from the left end of the resultant value, a *size* (hex 0C0A) exception is signaled. An attempt to extract the absolute value of a maximum negative binary value into a binary scalar of the same size also results in a *size* (hex 0C0A) exception.

For floating-point operations that involve a fixed-point receiver field, if nonzero digits would be truncated from the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point receiver operand, if the exponent of the resultant value is either too large or too small to be represented in the *receiver*, the *floating-point overflow* (hex 0C06) exception or the *floating-point underflow* (hex 0C07) exception is signaled.

Resultant Conditions:

•

- Positive-The algebraic value of the *receiver* operand is positive.
- Zero-The algebraic value of the *receiver* operand is zero.
- Unordered-The value assigned a floating-point *receiver* operand is NaN.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Find Byte (FINDBYTE)

Bound program access

```
Built-in number for FINDBYTE is 20.
FINDBYTE (
          source_string      : address of aggregate(*)
          search_character   : signed binary(4) - rightmost byte specifies
                                                  the search character  OR
                               unsigned binary(1)  OR
                               aggregate(1)
) :   space pointer(16) to the first character in the string that matches the
      search character
```

Description: The string specified by *source string* is searched for the value specified by *search character.* The search terminates when the value is found and a space pointer to the character is returned.

The results are undefined if the *source string* does not contain the *search character.*

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Find Character Constrained (MEMCHR)

Bound program access

```
Built-in number for MEMCHR is 22.
MEMCHR (
        source_string      : address of aggregate(*)
        search_character   : signed binary(4) - rightmost byte specifies
                                                the search character  OR
                             unsigned binary(1)  OR
                             aggregate(1)
        maximum_length     : unsigned binary(4) value which specifies the
                             maximum number of characters to search
) :   space pointer(16) to the first character in the string that matches the
      search character.  If the character is not found, null pointer value is
      returned
```

Description: The string specified by *source string* is searched for the value specified by *search character*. The search terminates if the value is found or the number of characters specified by *maximum length* have been searched. If the value is found, a space pointer to the character is returned. Otherwise, a null pointer value is returned.

If *maximum length* is 0, a null pointer value is returned.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Find Independent Index Entry (FNDINXEN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0494	Receiver	Index	Option list	Search argument

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: Space pointer.

Operand 4: Space pointer.

Bound program access

```
Built-in number for FNDINXEN is 36.
FNDINXEN (
          receiver          : address
          index             : address of system pointer
          option_list       : address
          search_argument   : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: Search the independent index identified by *index* (operand 2) according to the search criteria specified in the *option list* (operand 3) and the *search argument* (operand 4); then return the desired entry or entries in the *receiver* operand (operand 1). The maximum size of the independent index entry is either 120 bytes or 2,000 bytes depending on how the *maximum entry length* attribute field was specified when the index was created. Note that all indexes created in Version 3 Release 6 or later have a maximum entry length of 2,000 bytes.

The *option list* is a variable-length area that identifies the type of search to be performed, the length of the search argument(s), the maximum number of entries to be returned, the number of entries returned, the length of each entry returned, and the offsets to the entries within the receiver identified by the *receiver* (operand 1) space pointer. The *option list* has the following format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Rule option	Char(2)	
2	2	Argument length	UBin(2)	
4	4	Argument offset	Bin(2)	
6	6	Occurrence count	Bin(2)	
8	8	Return count	Bin(2)	
10	A	Returned index entry	[*] Char(4)	
		(Repeated *return count* times)		
10	A		Entry length	UBin(2)
12	C		Offset	Bin(2)
*	*	— End —		

The **rule option** identifies the type of search to be performed and has the following meaning:

Search Type	Value (Hex)	Meaning
=	0001	Find equal occurrences of operand 4. This option will return entries that match the search argument or begin with the search argument.
>	0002	Find occurrences that are greater than operand 4.
<	0003	Find occurrences that are less than operand 4.
>=	0004	Find occurrences that are greater than or equal to operand 4.

Search Type	Value (Hex)	Meaning
<=	0005	Find occurrences that are less than or equal to operand 4.
First	0006	Find the first index entry or entries.
Last	0007	Find the last index entry or entries.
Between	0008	Find all entries between the two arguments specified by operand 4 (inclusive). With this option, entries that match either search argument or begin with either search argument will be included.

The *rule option* to find *between* requires that operand 4 be a 2-element array in which element 1 is the starting argument and element 2 is the ending argument. All arguments between (and including) the starting and ending arguments are returned, but the *occurrence count* specified is not exceeded.

If the index was created to contain *both pointers and scalar data*, then the *search argument* must be 16-byte aligned. For the option to find between limits, both search arguments must be 16-byte aligned.

The *rule option* and the *argument length* determine the search criteria used for the index search. The *argument length* must be greater than or equal to one. The *argument length* for fixed-length entries must be less than or equal to the *argument length* specified when the index is created.

The **argument length** input field specifies the length of the *search argument* (operand 4) to be used for the index search. When the *rule option* equals *first* or *last*, the *argument length* field is ignored. For the *rule option* to find *between*, the *argument length* field specifies the length of one array element. The lengths of the array elements must be equal.

The **argument offset** input field specifies the offset of the second search argument from the beginning of the entire *search argument* field (operand 4). The *argument offset* field is ignored unless the *rule option* is find *between*.

The **occurrence count** input field specifies the maximum number of index entries that satisfy the search criteria to be returned. This field is limited to a maximum value of 4,095. If this value is exceeded, a *template value invalid* (hex 3801) exception is signaled.

The **return count** output field specifies the number of index entries satisfying the search criteria that were returned in the *receiver* (operand 1). If this field is 0, no index arguments satisfied the search criteria.

There are two output fields in the option list for each entry returned in the *receiver* (operand 1). The **entry length** is the length of the entry retrieved from the index. The **offset** has the following meaning:

-
- For the first entry, the *offset* is the number of bytes from the beginning of the *receiver* (operand 1) to the first byte of the first entry.
- For any succeeding entry, the *offset* is the number of bytes from the beginning of the immediately preceding entry to the first byte of the entry returned.

The entries that are retrieved as a result of the Find Independent Index Entry instruction are always returned starting with the entry that is closest to or equal to the *search argument* and then proceeding away from the *search argument*. For example, a search that is for < (less than) or <= (less than or equal to) returns the entries in order of decreasing value.

All the entries that satisfy the search criteria (up to the *occurrence count*) are returned in the space starting at the location designated by the *receiver* (operand 1) space pointer.

If the index was created to contain *both pointers and scalar data*, then each returned entry is 16-byte aligned.

If the index was created to contain *scalar data only*, then returned entries are contiguous.

Every entry retrieved causes the count of the find operations to be incremented by 1. The current value of this count is available through the Materialize Independent Index Attributes (MATINXAT) instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

- Retrieve
  –
  – Operand 2
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Operand 2
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3802 Template Size Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Find Relative Invocation Number (FNDRINVN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0543	Relative invocation number	Search range	Search criterion template

Operand 1: Signed binary(4) variable scalar.

Operand 2: Character(48) scalar or null.

Operand 3: Space pointer.

Bound program access

```
Built-in number for FNDRINVN is 124.
FNDRINVN (
          relative_invocation_number   : address of signed binary(4)
          search_range                 : address OR
                                         null operand
          search_criterion_template    : address
)
```

Note

It is recommended that you use search options 8, 9 and 10 for 8-byte invocation, activation and activation group marks, respectively, rather than search options 4, 5 and 6. 4-byte marks can wrap and produce unexpected results.

Description: The invocations identified by operand 2 are searched in the order specified by operand 2 until an invocation is found which satisfies the search criterion specified in the operand 3 template. The identity of the first invocation (in search order) to satisfy the search criterion is returned in operand 1. If no invocation in the specified range satisfies the search criterion, then either an exception is signaled, or a value of zero is returned in operand 1, depending on the modifiers specified in the operand 3 template.

Operand 1 is returned as a signed binary(4) value identifying the first invocation found that satisfies the specified search criterion. It is specified relative to the starting invocation identified by operand 2. A positive number indicates a displacement in the direction of newer invocations, while a negative number indicates a displacement in the direction of older invocations. A zero value can either indicate that no invocation in the specified range matched the specified criterion, or the starting invocation satisfied the specified criterion, depending on the modifiers specified in the operand 3 template. Operand 1 is not modified in the event that the instruction terminates with an exception.

Note that a modifier in the operand 3 template determines if the starting invocation identified by operand 2 is to be skipped. If the starting invocation is specified to be skipped during the search then a result of zero in operand 1 indicates failure to find an invocation that satisfies the criterion. If the starting invocation is specified not to be skipped, then a result of zero indicates the starting invocation has satisfied the specified criterion. If the starting invocation is specified not to be skipped and no invocation is found that satisfies the search criterion, an exception will be signaled.

Operand 2 identifies the starting invocation and the range of the search. If operand 2 is specified as a null operand, then operand 2 is assumed to identify a range starting with the current invocation and proceeding through all existing older invocations.

Operand 3 is a space pointer to a template that identifies the search criterion and search modifiers for the find operation.

Operand 2: The value specified by operand 2 identifies the range of invocations to be searched. This operand can be null (which indicates the range which starts with the current invocation and proceeds

through all existing older invocations), or it can contain either an invocation pointer to an invocation or a null pointer (which indicates a range starting with the current invocation).

Operand 2 has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Starting invocation offset	Bin(4)
4	4	Originating invocation offset (ignored)	Bin(4)
8	8	Invocation range	Bin(4)
12	C	Reserved (binary 0)	Char(4)
16	10	Starting invocation pointer	Invocation pointer
32	20	Reserved (binary 0)	Char(16)
48	30	—- End —-	

If a non-null pointer is specified for *starting invocation pointer*, then operand 2 must be 16-byte aligned in the space.

Terminology:

Requesting invocation The invocation executing the FNDRINVN instruction. Note that, in many cases, this invocation belongs to a system or language run-time procedure/program, and the instruction is actually being executed on behalf of another procedure or program.

Starting invocation The invocation which serves as the starting point for the search.

Field descriptions:

Starting invocation offset

A signed numerical value indicating an invocation relative to the invocation located by the *starting invocation pointer*. A value of zero denotes the invocation addressed by the *starting invocation pointer*, with increasingly positive numbers denoting increasingly later invocations in the stack, and increasingly negative numbers denoting increasingly earlier invocations in the stack.

If the *starting invocation pointer* is valid or null, but the invocation identified by this offset does not exist in the stack, an *invocation offset outside range of current stack* (hex 2C1A) exception will be signaled.

Originating invocation offset

This field is used by other instructions but is ignored by FNDRINVN.

Invocation range

Invocation range is a signed numerical value which specifies the direction of the search and the maximum number of invocations to be examined. The magnitude of *invocation range* specifies the maximum number of invocations to be searched *exclusive* of the starting invocation. It is not an error if this magnitude is greater than the number of existing invocations in the specified direction. If the sign of *invocation range* is positive (and non-zero), the search is performed in the direction of newer invocations, while if the sign is negative, the search is performed in the direction of older invocations.

Note that the *bypass starting invocation* modifier in operand 3 affects how the starting invocation is treated. If this modifier is binary 0, then the starting invocation is the first invocation examined. If *invocation range* is zero in this case then *only* the starting invocation is examined. If, on the other hand, *bypass starting invocation* is binary 1, then the starting invocation *does not* participate in the search, and, if *invocation range* is zero, no invocations are searched and a value of zero is returned for operand 1.

Starting invocation pointer

An invocation pointer to an invocation. If null, then the current invocation is indicated. If not null, then operand 2 must be 16-byte aligned in the space.

If the pointer identifies an invocation in another thread, a *process object access invalid* (hex 2C11) exception will be signaled. If the invocation identified by this pointer does not exist in the stack, an *object destroyed* (hex 2202) exception will be signaled.

Usage note: In cases where *starting invocation pointer* is null, operand 2 may be a constant.

Operand 3: The *search criterion template* identified by operand 3 must be aligned on a 16-byte boundary. The template is a 32-byte value with the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Reserved (binary 0)	Char(8)		
8	8	Search option	Bin(4)		
12	C	Search modifiers	Char(4)		
12	C		Bypass starting invocation		Bit 0
			0 =	The starting invocation identified by operand 2 is the first invocation tested. An *invocation not found* (hex 1E02) exception is signaled if the search criterion is not satisfied.	
			1 =	The starting invocation identified by operand 2 is skipped and no exception is signaled if the search criterion is not satisfied.	
12	C		Compare for mismatch		Bit 1
			0 =	The instruction identifies the first invocation (in specified search order) which *matches* the specified search criterion	
			1 =	The instruction identifies the first invocation (in specified search order) which *does not match* the specified search criterion	
12	C		Reserved (binary 0)		Bits 2-31
16	10	Search argument	Char(16)		
32	20	—- End —-			

Search option

Specifies the invocation attribute to be examined:

1 Routine type. *Search argument* is a one-byte routine type, left aligned. Allowed *search argument* values are:

Hex 01 =
Non-Bound Program

Hex 02 =
Bound Program Entry Procedure (PEP)

Hex 03 =
Bound Program Procedure

Note: Bound program procedures are contained within bound programs, bound service programs, and Java(TM) programs. All discussion of bound program procedure semantics also apply to Java program procedures.

2 Invocation type. *Search argument* is a one-byte invocation type, left aligned. Allowed *search argument* values are:

Hex 01 =
Call external

Hex 02 =
Transfer control

Hex 03 =
Event handler

Hex 04 =
External exception handler (for non-bound program)

Hex 05 =
Initial program in process problem state

Hex 06 =
Initial program in process initiation state

Hex 07 =
Initial program in process termination state

Hex 08 =
Invocation exit (for non-bound program)

Hex 09 =
Return or return/XCTL trap handler

Hex 0A =
Call program

Hex 0B =
Cancel handler (bound program only)

Hex 0C =
Exception handler (bound program only)

Hex 0D =
Call bound procedure/call with procedure pointer

Hex 0E =
Process Default Exception Handler

3 Invocation status. *Search argument* consists of two four-byte fields, left aligned. The invocation status of each examined invocation is ANDed with the first field and then compared to the second field.

4
Invocation mark. *Search argument* is a four-byte invocation mark, left aligned. If the search is in the direction of older invocations, the result identifies the first invocation found with an invocation mark less than or equal to the search argument. If the search is in the direction of newer invocations, the result identifies the first invocation found with an invocation mark greater than or equal to the search argument. If *invocation range* is zero, then the search is satisfied only if the invocation mark of the *starting invocation* exactly matches the search argument, and this can occur only if *bypass starting invocation* is binary 0.

For this option *compare for mismatch* is ignored.

5
Activation mark. *Search argument* is a four-byte activation mark, left aligned. The activation mark of the program or module activation corresponding to each examined invocation is compared to *search argument*. Invocations with no activation (ie, the invocations of non-bound reentrant programs, and the invocation stack base entry) are considered to have an activation mark of binary 0.

6
Activation group mark. *Search argument* is a four-byte activation group mark, left aligned. The activation group mark of each examined invocation is compared to *search argument*. The activation group mark of each examined invocation is determined from the activation associated with the invocation. (Each activation belongs to a single activation group.) However,

- if no activation exists for the invocation, or
- if an activation exists and it belongs to an shared activation group owned by another process

then, the activation group mark for the examined invocation is taken to be,

1 for a system-state invocation

2 for a user-state invocation

7
Program pointer. *Search argument* is a system pointer to a program. The program corresponding to each examined invocation is compared to the program identified by the pointer.

8
Invocation mark. *Search argument* is an eight-byte invocation mark, left aligned. If the search is in the direction of older invocations, the result identifies the first invocation found with an invocation mark less than or equal to the search argument. If the search is in the direction of newer invocations, the result identifies the first invocation found with an invocation mark greater than or equal to the search argument. If *invocation range* is zero, then the search is satisfied only if the invocation mark of the *starting invocation* exactly matches the search argument, and this can occur only if *bypass starting invocation* is binary 0.

For this option *compare for mismatch* is ignored.

9
Activation mark. *Search argument* is an eight-byte activation mark, left aligned. The activation mark of the program or module activation corresponding to each examined invocation is compared to *search argument*. Invocations with no activation (ie, the invocations of non-bound reentrant programs, and the invocation stack base entry) are considered to have an activation mark of binary 0.

10
Activation group mark. *Search argument* is an eight-byte activation group mark, left aligned. The activation group mark of each examined invocation is compared to *search argument*. The activation group mark of each examined invocation is determined from the activation associated with the invocation. (Each activation belongs to a single activation group.) However,

- if no activation exists for the invocation, or
- if an activation exists and it belongs to a shared activation group owned by another process

then, the activation group mark for the examined invocation is taken to be,

1 for a system-state invocation

2 for a user-state invocation

Bypass starting invocation
If *bypass starting invocation* is binary 0, then the starting invocation specified by operand 2 is the first invocation examined. In this case, if the *invocation range* of operand 2 is exhausted without satisfying the search criterion then a *template value invalid* (hex 3801) exception is signaled, with the *search argument* field of operand 3 identified as the erroneous field.

If *bypass starting invocation* is binary 1, then the starting invocation specified by operand 2 is skipped, and a failure to satisfy the search criterion is indicated by returning a binary 0 value in operand 1.

Compare for mismatch
If *compare for mismatch* is binary 0, then the search criterion is satisfied when an invocation is found whose attribute *matches* the *search argument*. If *compare for mismatch* is binary 1, however, then the search criterion is satisfied when an invocation is found whose attribute *does not match* the *search argument*.

Search argument
A value of between one and 16 bytes as described above. Unused bytes are ignored.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

16 Exception Management

1603 Invalid Invocation Address

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1E Machine Observation

1E02 Invocation Not Found

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C11 Process Object Access Invalid

2C1A Invocation Offset Outside Range of Current Stack

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Free Activation Group-Based Heap Space Storage (FREHSS)

Op Code (Hex)	Operand 1
03B5	Space allocation

Operand 1: Space pointer.

Bound program access

```
Built-in number for FREHSS is 114.
FREHSS (
        space_allocation   : address
)
```

Note: The term ″heap space″ in this instruction refers to an ″activation group-based heap space″.

Description: The heap space storage allocation beginning at the byte addressed by operand 1 is de-allocated from the heap space which supplied the allocation. De-allocation makes the storage available for reuse by subsequent Allocate Activation Group-Based Heap Space Storage (ALCHSS) instructions. The entire space allocation is de-allocated; partial de-allocation is not supported. A free of heap space storage can be performed without regard to the activation group in which it was allocated, as long as the allocation was done by a thread in the same process.

Operand 1 must be exactly equal to the space pointer that was returned by some previous Allocate Activation Group-Based Heap Space Storage (ALCHSS) or Reallocate Activation Group-Based Heap Space Storage (REALCHSS) instruction. If it is not, an *invalid request* (hex 4502) exception will be signaled.

Subsequent references to space allocations which have been freed cause unpredictable results.

FREHSS will signal an *object domain or hardware storage protection violation* (hex 4401) exception if a program running user state attempts to de-allocate heap space storage in a heap space with a domain of *system.*

Operand 1 is not modified by the instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

45 Heap Space

4502 Invalid Request

4505 Heap Space Destroyed

4506 Invalid Heap Space Condition

Free Activation Group-Based Heap Space Storage From Mark (FREHSSMK)

Op Code (Hex)	**Operand 1**
03B9	Mark identifier

Operand 1: Space pointer data object.

Bound program access
Built-in number for FREHSSMK is 115. FREHSSMK (mark_identifier : address of space pointer(16))

Note: The term ″heap space″ in this instruction refers to an ″activation group-based heap space″.

Description: All heap space allocations which have occurred from the heap space since it was marked, with the *mark identifier* supplied in operand 1, are freed. This may include heap space storage marked by intervening Set Activation Group-Based Heap Space Storage Mark (SETHSSMK) instructions. The *mark identifier* specified in operand 1 and all mark identifiers obtained since the heap space was marked by operand 1 are cleared from the heap space. An attempt to free heap space storage from a mark that has already been cleared by a previous FREHSSMK instruction will result in an *invalid mark identifier* (hex 4507) exception. A free of heap space storage can be performed without regard to the activation group in which it was allocated, as long as the allocation was done by a thread in the same process.

FREHSSMK will signal an *object domain or hardware storage protection violation* (hex 4401) exception if a program running user state attempts a Free Activation Group-Based Heap Space Storage From Mark for a heap space with a *domain* of *system*.

Operand 1 is not modified by the instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C09 Auxiliary Storage Pool Number Invalid

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

45 Heap Space

4502 Invalid Request

4505 Heap Space Destroyed

4506 Invalid Heap Space Condition

4507 Invalid Mark Identifier

Generate Universal Unique Identifier (GENUUID)

Op Code (Hex)
Operand 1
011D
UUID return template
Operand 1: **Space pointer.**

Bound program access
Built-in number for GENUUID is 461. GENUUID (UUID_return_template : address)

Description: **This instruction generates a universal unique identifier and returns it in the template provided. The UUID is unique as an identifier across all time and space and is consistent with the Open Systems Foundation (OSF) Distributed Computing Environments (DCE) version 1 UUID specification described in the DCE's "Architecture Environment Specification/Distributed Computing: for Remote Procedure Calls", Appendix A. The template identified by operand 1 must be 16 byte aligned. The 16 byte Universal Unique Identifier (UUID) is returned in the** *UUID return template*.

The *UUID return template* (operand 1) has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Return template size specification	Char(8)	
0	0		Number of bytes provided	UBin(4)
4	4		Number of bytes available	UBin(4)
8	8	Reserved (binary 0)	Char(8)	
16	10	Returned UUID	Char(16)	
32	20	—- End —-		

The first 4 bytes of the template identify the total number of bytes provided for use by the instruction. The value is supplied as input to the instruction and is not modified by the instruction. A value of less than 32 causes a *template size invalid* (hex 3802) exception to be signaled.

The second 4 bytes of the template identify the total number of bytes available to be returned. This value is output. The value is the size of the template.

The reserved field must be set to zeros or a *template value invalid* (hex 3801) exception will be signaled.

The returned UUID field contains the generated UUID. The UUID is a DCE version 1 UUID.

Note: There are certain restrictions on the UUID that should be remembered:

-
- **Operations on the UUID should be limited to:**
  -
  - **Equality comparison (equal or not equal)**
  - **Lexical ordering - UUIDs can be ordered based on the most significant byte that differs between the compared UUIDs.**
  - **String conversion - based on the DCE architecture string representation of a UUID.**

Warning: Temporary Level 3 Header

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

32 Scalar Specification

3201 Scalar Type Invalid

38 Template Specification

3801 Template Value Invalid

3802 Template Size Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Increment Date (INCD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0404	Result date	Source date	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

Bound program access

```
Built-in number for INCD is 95.
INCD (
      result_date            : address
      source_date            : address
      duration               : address of packed decimal
      instruction_template   : address
)
```

Description: The date specified by operand 2 is incremented by the date *duration* specified by operand 3. The resulting date is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the *instruction template*.

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Instruction template	Char(*)		
0	0		Instruction template size	Bin(4)	
4	4		Operand 1 data definitional attribute template number	UBin(2)	
6	6		Operand 2 data definitional attribute template number	UBin(2)	
8	8		Operand 3 data definitional attribute template number	UBin(2)	
10	A		Operand 1 length	UBin(2)	
12	C		Operand 2 length	UBin(2)	
14	E		Operand 3 length	UBin(2)	
14	E		Fractional number of digits		Char(1)
15	F		Total number of digits		Char(1)
16	10		Input indicators	Char(2)	
16	10		End of month adjustment		Bit 0
			0 = No adjustment		
			1 = Adjustment		
16	10		Tolerate data decimal errors		Bit 1
			0 = No toleration		
			1 = Tolerate		
16	10		Reserved (binary 0)		Bits 2-1
18	12		Output indicators	Char(2)	
18	12		End of month adjustment		Bit 0
			0 = No adjustment		
			1 = Adjustment		
18	12		Reserved (binary 0)		Bits 1-1
20	14		Reserved (binary 0)	Char(22)	
42	2A		Data definitional attribute template list	Char(*)	
42	2A		Size of the DDAT list		UBin(4
46	2E		Number of DDATs		UBin(2
48	30		Reserved (binary 0)		Char(1
58	3A		DDAT offset		[*] UBi
*	*		Data definitional attribute template		[*] Cha
*	*	—- End —-			

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a date and identical. The DDAT for operand 3 must be valid for a date duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length, **operand 2 length**, and **operand 3 length** are specified in number of bytes.

The input indicator, **end of month adjustment**, is used to allow or disallow the occurrence of an end of month adjustment.

The input indicator, **tolerate data decimal errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The output indicator, **end of month adjustment**, is used to indicate an end of month adjustment, when end of month adjustments are allowed.

End of month adjustment is the following concept. For SAA(R), the result of adding a 1 month duration to the date 01/31/1989 is 02/28/1989. The days portion is adjusted to fit the month, 31 is changed to 28. When this happens,the *end of month adjustment* output indicator is set to *adjustment*.

When *end of month adjustments are not allowed*, the month and year definitions in the data definition attribute template must have values greater than zero, otherwise a *template value invalid* (hex 3801) exception will be signaled. The result of adding a 1 month duration to the Gregorian date 01/31/1989 is 03/02/1989, when the definition of a month is 30 days.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes definitional attributes of the operands. The length of the date and date duration character operands will be defined by the template. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C15 Date Boundary Overflow

0C16 Data Format Error

0C17 Data Value Error

0C18 Date Boundary Underflow

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Increment Time (INCT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0434	Result time	Source time	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

Bound program access

```
Built-in number for INCT is 97.
INCT (
      result_time            : address
      source_time            : address
      duration               : address of packed decimal
      instruction_template   : address
)
```

Description: The time specified by operand 2 is incremented by the time *duration* specified by operand 3. The resulting time is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the *instruction template*.

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Instruction template	Char(*)	
0	0		Instruction template size	Bin(4)
4	4		Operand 1 data definitional attribute template number	UBin(2)
6	6		Operand 2 data definitional attribute template number	UBin(2)
8	8		Operand 3 data definitional attribute template number	UBin(2)
10	A		Operand 1 length	UBin(2)
12	C		Operand 2 length	UBin(2)
14	E		Operand 3 length	UBin(2)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
14	E		Fractional number of digits	Char(1)
15	F		Total number of digits	Char(1)
16	10		Input indicators	Char(2)
16	10		Reserved (binary 0)	Bit 0
16	10		Tolerate data decimal errors 0 = No toleration 1 = Tolerate	Bit 1
16	10		Reserved (binary 0)	Bits 2-15
18	12		Reserved (binary 0)	Char(24)
42	2A		Data definitional attribute template list	Char(*)
42	2A		Size of the DDAT list	UBin(4)
46	2E		Number of DDATs	UBin(2)
48	30		Reserved (binary 0)	Char(10
58	3A		DDAT offset	[*] UBin
*	*		Data definitional attribute template	[*] Char
*	*	—- End —-		

A **data definitional attribute template number (DDAT)** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a time and identical. The DDAT for operand 3 must be valid for a time duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length, **operand 2 length**, and **operand 3 length** are specified in number of bytes.

The input indicator, **tolerate data decimal errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the time and time duration character operands will be defined by the templates. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C16 Data Format Error

0C17 Data Value Error

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Increment Timestamp (INCTS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
040C	Result timestamp	Source timestamp	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

Bound program access

```
Built-in number for INCTS is 99.
INCTS (
       result_timestamp       : address
       source_timestamp       : address
       duration               : address of packed decimal
       instruction_template   : address
)
```

Description: The timestamp specified by operand 2 is incremented by the date, time, or timestamp *duration* specified by operand 3. The resulting timestamp is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the *instruction template*.

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Instruction template	Char(*)		
0	0		Instruction template size	Bin(4)	
4	4		Operand 1 data definitional attribute template number	UBin(2)	
6	6		Operand 2 data definitional attribute template number	UBin(2)	
8	8		Operand 3 data definitional attribute template number	UBin(2)	
10	A		Operand 1 length	UBin(2)	
12	C		Operand 2 length	UBin(2)	
14	E		Operand 3 length	UBin(2)	
14	E		Fractional number of digits		Char(1)
15	F		Total number of digits		Char(1)
16	10		Input indicators	Char(2)	
16	10		End of month adjustment		Bit 0
			0 = No adjustment		
			1 = Adjustment		
16	10		Tolerate data decimal errors		Bit 1
			0 = No toleration		
			1 = Tolerate		
16	10		Reserved (binary 0)		Bits 2-15
18	12		Output indicators	Char(2)	
18	12		End of month adjustment		Bit 0
			0 = No adjustment		
			1 = Adjustment		
18	12		Reserved (binary 0)		Bits 1-15
20	14		Reserved (binary 0)	Char(22)	
42	2A		Data definitional attribute template list	Char(*)	
42	2A		Size of the DDAT list		UBin(4)
46	2E		Number of DDATs		UBin(2)
48	30		Reserved (binary 0)		Char(10)
58	3A		DDAT offset		[*] UBin(4)
*	*		Data definitional attribute template		[*] Char(*)
*	*	— End —			

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example, the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a timestamp and identical. The DDAT for operand 3 must be valid for a timestamp duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length, **operand 2 length**, and **operand 3 length** are specified in number of bytes.

The **input indicator, end of month adjustment**, is used to allow or disallow the occurrence of an end of month adjustment.

The input indicator, **tolerate decimal data errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The **output indicator, end of month adjustment**, is used to indicate an end of month adjustment, when end of month adjustments are allowed.

End of month adjustment is the following concept. For SAA(R), the result of adding a 1 month duration to the date 01/31/1989 is 02/28/1989. The days portion is adjusted to fit the month, 31 is changed to 28. When this happens, the *end of month adjustment output indicator* is set to *on*.

When *end of month adjustments are not allowed*, the month and year definitions in the data definition attribute template must have values greater than zero, otherwise a *template value invalid* (hex 3801) exception will be signaled. The result of adding a 1 month duration to the Gregorian date 01/31/1989 is 03/02/1989, when the definition of a month is 30 days.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the timestamp and duration character operands will be defined by the template. For a further description of the data definitional attribute template, see Data Definitional Attribute Template.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C15 Date Boundary Overflow

0C16 Data Format Error

0C17 Data Value Error

0C18 Date Boundary Underflow

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Initialize Exception Handler Control Actions (INITEHCA)

Bound program access

`Built-in number for INITEHCA is 384.`
`INITEHCA (`
`)`

Description: The ″invocation-class″ control action fields associated with exception handlers scoped to the current procedure are initialized.

The initial value of each control action field is defined by the exception handler dictionary entry associated with the handler.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

Insert Independent Index Entry (INSINXEN)

Op Code (Hex)	**Operand 1**	**Operand 2**	**Operand 3**
04A3	Index	Argument	Option list

Operand 1: System pointer.

Operand 2: Space pointer.

Operand 3: Space pointer.

Bound program access
Built-in number for INSINXEN is 37. INSINXEN (index : address of system pointer argument : address option_list : address)

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: Insert one or more new entries into the independent index identified by operand 1 according to the criteria specified in the *option list* (operand 3). Each entry is inserted into the index at the appropriate location based on the binary value of the *argument*. No other collating sequence is supported. The maximum length allowed for the independent index entry is either 120 bytes or 2,000 bytes depending on how the *maximum entry length* attribute field was specified when the index was created. Note that all indexes created in Version 3 Release 6 or later have a maximum entry length of 2,000 bytes.

The *argument* (operand 2) and the *option list* (operand 3) have the same format as the search argument and option list for the Find Independent Index Entry (FNDINXEN) instruction.

The **rule option** identifies the type of insert to be performed and has the following meaning:

Insert Type	Value (Hex)	Meaning	Authorization
Insert	0001	Insert unique argument	Insert
Insert with replacement	0002	Insert argument, replacing the nonkey portion if the key is already in the index	Update
Insert without replacement	0003	Insert argument only if the key is not already in the index	Insert

The *insert rule option* is valid only for indexes not containing keys. The *insert with replacement rule option* and the *insert without replacement rule option* are valid for indexes containing either fixed- or variable-length entries with keys. The *duplicate key argument in index* (hex 1801) exception is signaled for the following conditions:

-
- If the *rule option* is *insert* and the argument to be inserted (operand 2) is already in the index
- If the *rule option* is *insert without replacement* and the key portion of the argument to be inserted (operand 2) is already in the index

The *argument length* and *argument offset* fields are ignored, however, the *entry length* and *offset* fields must be entered for every entry which is to be inserted into the index.

The **occurrence count** specifies the number of arguments to be inserted. This field is limited to a maximum value of 4,095. If this value is exceeded, a *template value invalid* (hex 3801) exception is signaled.

If the index was created to contain *both pointers and scalar data*, then each entry to be inserted must be 16-byte aligned. If the index was created to contain *variable-length* entries, then the *entry length* and *offset fields* must be specified in the *option list* for each *argument* in the space identified by operand 2. The *entry length* is the length of the entry to be inserted.

If the index was created to contain *both pointers and scalar data*, the *offset* field in the *option list* must be supplied for each entry to be inserted. The *offset* is the number of bytes from the beginning of the previous entry to the beginning of the entry to be inserted. For the first entry, this is the offset from the start of the space identified by operand 2.

The *return count* specifies the number of entries inserted into the index. If the index was created to contain *scalar data only*, then any pointers inserted are invalidated.

Warning: Temporary Level 3 Header

Authorization Required

•

• Insert or update depending on insert type
  –
  – Operand 1
• Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize
  –
  – Contexts referenced for address resolution
• Modify
  –
  – Operand 1

Exceptions

02 Access Group

0201 Object Ineligible for Access Group

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

18 Independent Index

1801 Duplicate Key Argument in Index

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C0E IASP Resources Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3802 Template Size Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Invocation Pointer (INVP)

Bound program access

```
Built-in number for INVP is 6.
INVP (
      relative_invocation   : unsigned binary(4) literal value which
                              specifies the relative invocation
) :   invocation pointer which references the invocation specified by relative_invocation
```

Description: The *relative invocation* is a literal value between 0 and scope-1 where scope is the procedure's scoping level. A value of 0 indicates that an invocation pointer to the currently executing procedure is to be returned. A value of 1 indicates that an invocation pointer for the most recent invocation of the procedure identified as the owner of the currently executing procedure is returned. Increasing values for *relative invocation* refer to the further static nesting of the currently executing procedure, where invocation pointers to the most recent invocations of those uplevel procedures are returned.

The invocation pointer is only valid while the procedure invocation referenced by it still exists.

Restriction
The *relative invocation* must be 0. Values greater than 0 are currently not supported.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

Lock Object (LOCK)

Op Code (Hex)	Operand 1
03F5	Lock request template

Operand 1: Space pointer.

Bound program access
Built-in number for LOCK is 46. LOCK (lock_request_template : address)

Description: Locks for system objects identified by system pointers in the space identified by operand 1 are allocated to the requesting thread or its containing process or a transaction control structure. The lock state desired for each object is specified by a value associated with each system pointer in the *lock request template* (operand 1).

The *lock request template* must be aligned on a 16-byte boundary. The format is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Number of lock requests in template	Bin(4)
4	4	Offset to lock state selection values	Bin(2)
6	6	Wait time-out value for instruction	Char(8)
14	E	Lock request options	Char(2)
14	E		Lock request type **00 =** Immediate request- If all locks cannot be immediately granted, signal *lock request not grantable* (hex 1A02) exception. **01 =** Synchronous request- Wait until all locks can be granted. **10 =** Asynchronous request- Allow processing to continue and signal event when the object is available.
14	E		Access state modifications
14	E		When the thread is entering lock wait for synchronous **0 =** Access state should not be modified. **1 =** Access state should be modified.
14	E		When the thread is leaving lock wait: **0 =** Access state should not be modified. **1 =** Access state should be modified.
14	E		Reserved (binary 0)
14	E		Time-out option **0 =** Wait for specified time, then signal time-out exception. **1 =** Wait indefinitely.
14	E		Template extension specified

Offset Dec	Offset Hex	Field Name	Data Type and Length
			0 = Template extension is not specified.
			1 = Template extension is specified.
14	E		Lock scope
			0 = Lock is scoped to the *lock scope object type*.
			1 = Lock is scoped to the current thread.
14	E		Lock scope object type
			0 = Process containing the current thread.
			1 = Transaction control structure attached to the current thread.
14	E		Reserved (binary 0)
16	10	—- End —-	

The **lock object template extension** is only present if *template extension specified* is indicated above. Otherwise, the *object(s) to be locked* should immediately follow.

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Lock object template extension	Char(16)
0	0		Extension options
0	0		Modify thread event mask option
			0 = Do not modify thread event mask
			1 = Modify thread event mask
0	0		Asynchronous signals processing option
			0 = Do not allow asynchronous signal proces
			1 = Allow asynchronous signal processing du
0	0		Reserved (binary 0)
1	1		Extension area
1	1		New thread event mask
3	3		Previous thread event mask
5	5		Reserved (binary 0)
16	10	Object(s) to be locked	[*] System pointer
		This should be repeated as specified by *number of lock requests in template* above.	
*	*	—- End —-	

The *lock state selection* is located by adding the *offset to lock state selection values* above to operand 1.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Lock state selection	[*] Char(1)	
		(repeated for each pointer in the template)		
0	0		Requested lock state	Bits 0-4

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			(1 = lock requested, 0 = lock not requested) Only one state may be requested.	
0	0		LSRD lock	Bit 0
0	0		LSRO lock	Bit 1
0	0		LSUP lock	Bit 2
0	0		LEAR lock	Bit 3
0	0		LENR lock	Bit 4
0	0		Reserved (binary 0)	Bits 5-6 +
0	0		Entry active indicator	Bit 7
			0 = Entry not active - This entry is not used.	
			1 = Entry active - Obtain this lock.	
*	*	—- End —-		

Note: Fields indicated with a plus sign (+) are ignored by the instruction.

Lock Allocation Procedure: A single Lock Object instruction can request the allocation of one or more lock states on one or more objects. Locks are allocated sequentially until all locks requested are allocated.

When two or more threads are competing for a conflicting lock allocation on a system object, the machine attempts to first satisfy the lock allocation request of the thread with the highest priority. Within that priority, the machine attempts to satisfy the request that has been waiting longest.

If any exception is identified during the instruction's execution, any locks already granted by the instruction are released, and the lock request is canceled.

For each system object lock, counts are kept by lock state and by thread, process, or transaction control structure. When a lock request is granted, the appropriate lock count(s) of each lock state specified is incremented by 1.

If a transfer of a lock from another thread causes a previously unsatisfied lock request to become satisfied, the lock request and the transfer lock are treated independently relative to lock accounting. The appropriate lock counts are incremented for both the lock request and the transfer lock function.

The **offset to lock state selection values** specifies an offset from the beginning of the lock request. This offset is used to locate the lock state selection values.

The **wait time-out value for instruction** field establishes the maximum amount of time that a thread competes for the requested set of locks when *lock request type* is either *synchronous* or *asynchronous*. When *lock scope object type* has a value of binary 1, the lock wait time interval value for the transaction control structure attached to the current thread is used to establish the maximum amount of time that the thread competes for the set of locks. See "Standard Time Format" on page 1272 for additional information on the format of the *wait time-out value for instruction*.

The maximum wait time-out interval allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum wait time-out causes the maximum wait time-out to be used. If the *wait time-out* field is specified with a value of binary 0, then the value associated with the default wait time-out field in the process definition template establishes the time interval.

When a requested lock state cannot be immediately granted, any locks already allocated by this Lock Object instruction are released, and the **lock request type** specified in the lock request template establishes the machine action. The lock request types are described in the following paragraphs.

-
- *Immediate request*- If the requested locks cannot be granted immediately, this option causes the *lock request not grantable* (hex 1A02) exception to be signaled. No locks are granted and the lock request is canceled.
- *Synchronous request*- This option causes the thread requesting the locks to be placed in the wait state until all requested locks can be granted. If the locks cannot be granted in the time interval established by the *wait time-out* field specified in the lock request template, the *lock time-out* (hex 3A02) exception is signaled to the requesting thread at the end of the interval. No locks are granted, and the lock request is canceled.
- *Asynchronous request*- This option allows the requesting thread to proceed with execution while the machine asynchronously attempts to satisfy the lock request.

When:

- the thread requesting the locks is the only thread in the process at the time of the lock request, and
- the *synchronous request* option is specified, and
- the requested locks cannot be immediately allocated,

the **access state modification** field in the lock request template specifies whether the access state of the process access group is to be modified on entering and/or returning from the lock wait. The field has no effect if the process *instruction wait access state control attribute* specifies that no access state modification is allowed. If the process attribute value specifies that access state modification is allowed and the wait on event *access state modification* option specifies modify access state, the machine modifies the access state for the specified process access group.

If the thread requesting the locks belongs to a multi-threaded process, no access state modification is performed.

If the *lock request type* is *synchronous* and the invocation containing the lock instruction is terminated, then the lock request is canceled.

If the *lock request type* is *asynchronous* and the invocation containing the Lock Object instruction is terminated, then the lock request remains active.

The **lock scope** field and the **lock scope object type** field determines which scope all specified lock requests will be allocated to, either a thread, process or transaction control structure. When *lock scope* has a value of binary 0 and *lock scope object type* has a value of 0, the lock scope will be the process containing the current thread. When *lock scope* has a value of binary 0 and *lock scope object type* has a value of 1, the lock scope will be the transaction control structure that is attached to the current thread. If the current thread does not have a transaction control structure attached, then the lock scope will be the process containing the current thread. When *lock scope* has a value of binary 1 the lock scope will be to the current thread and the value of the *lock scope object type* will be used to determine how lock conflicts are detected. Locks scoped to a thread with a *lock scope object type* value of *process containing the current thread* can never conflict with a lock scoped to its containing process, but may conflict with a lock scoped to a different process, a transaction control structure, or any other thread (depending on the lock states involved). Locks scoped to a thread with a *lock scope object type* value of *transaction control structure attached to the current thread* can never conflict with a lock scoped to the transaction control structure, but may conflict with a lock scoped to a different transaction control structure, a process, or any other thread (depending on the lock states involved).

If *lock scope object type* has a value of *transaction control structure attached to the current thread* and the transaction control structure state does not allow objects to be locked on behalf of the transaction control structure, a *object not eligible for operation* (hex 2204) exception is signaled.

Allocated process scope locks are released when the process terminates. Allocated thread scope locks are released when the thread terminates. If a thread requested a process scope lock, the process will continue to hold that lock after termination of the requesting thread. If a thread requested a transaction control structure scope lock, the transaction control structure will continue to hold that lock after the termination of the requesting thread.

If a thread is terminated while waiting for a lock with a *lock request type* of either *synchronous* or *asynchronous*, the lock request is canceled regardless of the scope of the requested lock.

The **modify thread event mask option** controls the state of the event mask in the thread executing this instruction. If the event mask is in the *masked* state, the machine does not schedule signaled event monitors in the thread. The event monitors continue to be signaled by the machine or other threads. When the thread is modified to the *unmasked* state, event handlers are scheduled to handle those events that occurred while the thread was masked and those events occurring while in the unmasked state. The number of events retained while the thread is masked is specified by the attributes of the event monitor associated with the thread.

A lock request with an *asynchronous lock request type* cannot have the *modify thread event mask option* set to 1.

If the system security level machine attribute is hex 40 or greater and the thread is running in user state, then the *modify thread event mask option* is not allowed and a *template value invalid* (hex 3801) exception is signalled.

When the *modify thread event mask* is set to 1, the **previous thread event mask** will be returned and the **new thread event mask** will take effect only when the lock(s) have been successfully granted. If the lock request is not successful, the *previous thread event mask* value is not returned, nor does the *new thread event mask* take effect.

Whether masking or unmasking the current thread, the new mask takes effect upon completion of a satisfied lock object.

Valid masking values are:

0	Masked
256	Unmasked

Other values are reserved. If any other values are specified, a *template value invalid* (hex 3801) exception is signaled.

The thread is automatically masked by the machine when event handlers are invoked. If the thread is unmasked in the event handler, other events can be handled if another enabled event monitor within that thread is signaled. If the thread is masked when it exits from the event handler, the machine explicitly unmasks the thread.

The **asynchronous signals processing option** controls the action to be taken if an asynchronous signal is pending or received while in Lock wait. If an asynchronous signal that is not blocked or ignored is generated for the thread and the *asynchronous signals processing option* indicates *allow asynchronous signals processing during Lock wait*, the Lock wait will be terminated and an *asynchronous signal terminated MI wait* (hex 4C01) exception is signaled. Otherwise, when the *asynchronous signals processing option* indicates *do not allow asynchronous signals processing during Lock wait*, the thread remains in the wait until all requested locks can be granted or until the *wait time-out value for instruction* expires.

Warning: Temporary Level 3 Header

Authorization Required

•
- Some authority or ownership
  –
  – Objects to be locked
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•
- Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1A02 Lock Request Not Grantable

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C06 Machine Lock Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2204 Object Not Eligible for Operation
- 2207 Authority Verification Terminated Due to Destroyed Object
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

36 Space Management

- 3601 Space Extension/Truncation

38 Template Specification

- 3801 Template Value Invalid

3A Wait Time-Out

- 3A02 Lock Time-Out

44 Protection Violation

- 4401 Object Domain or Hardware Storage Protection Violation
- 4402 Literal Values Cannot Be Changed

4C Signals Management

- 4C01 Asynchronous Signal Terminated MI Wait

Lock Object Location (LOCKOL)

Op Code (Hex)	Operand 1
03C1	Lock request template

Operand 1: Space pointer.

Bound program access

```
Built-in number for LOCKOL is 498.
LOCKOL (
        lock_request_template   : address
)
```

Description: The instruction requests that the object locations identified in the lock request template (operand 1) be granted to the issuing thread.

The *lock request template* identified by operand 1 must be aligned on a 16-byte boundary. The format of the *lock request template* is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Number of object location lock requests in template	UBin(4)
4	4	Offset to lock state selection values	UBin(4)
8	8	Wait time-out value for instruction	Char(8)
16	10	Lock request options	Char(3)
16	10		Reserved (binary 0)
16	10		Lock request type **0 =** Immediate request-If all locks cannot be immediately granted, signal exception. **1 =** Synchronous request-Wait until all locks can be granted.
16	10		Access state modifications
16	10		When the process is entering lock wait for synchronous **0 =** Access state should not be modified. **1 =** Access state should be modified.
16	10		When the process is leaving lock wait: **0 =** Access state should not be modified. **1 =** Access state should be modified.
16	10		Reserved (binary 0)
16	10		Time-out option **0 =** Wait for specified time, then signal time-out exception. **1 =** Wait indefinitely.
16	10		Reserved (binary 0)
16	10		Asynchronous signals processing option **0 =** Do not allow asynchronous signal processing during Lock Object Location wait. **1 =** Allow asynchronous signal processing during Lock Object Location wait.
16	10		Reserved (binary 0)

Offset Dec	Offset Hex	Field Name	Data Type and Length
19	13	Reserved (binary 0)	Char(13)
32	20	Object location(s) to be locked	[*] Object pointer
		This should be repeated as specified by the *number of object location lock requests in template* field above.	
*	*	—- End —-	

The *lock state selection* field is located by adding the *offset to lock state selection values* above to operand 1.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Lock state selection	[*] Char(1)	
		(repeated for each object pointer in the template)		
0	0		Requested lock state	Bits 0-4
			(1 = lock requested, 0 = lock not requested)	
			Only one state may be requested; else the *template value invalid* (hex 3801) exception is signaled.	
0	0		LSRD lock	Bit 0
0	0		LSRO lock	Bit 1
0	0		LSUP lock	Bit 2
0	0		LEAR lock	Bit 3
0	0		LENR lock	Bit 4
0	0		Reserved (binary 0)	Bits 5-6
0	0		Entry active indicator	Bit 7
			0 = Entry not active- This entry is not used.	
			1 = Entry active- Obtain this lock.	
*	*	—- End —-		

Lock Allocation Procedure: A single Lock Object Location instruction can request the allocation of one or more locks on one or more object locations. Object location locks are granted sequentially until all the locks requested are granted.

The **wait time-out** field establishes the maximum amount of time that a thread competes for the requested set of locks when the *lock request type* is *synchronous*. See "Standard Time Format" on page 1272 for additional information on the format of the *wait time-out*. The maximum wait time-out interval allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum wait time-out causes the maximum wait time-out to be used. If the *wait time-out* field is specified with a value of binary 0, the default wait time-out for the process is used as the time interval.

When a requested lock state cannot be immediately granted, any locks already granted by this Lock Object Location instruction are released, and the *lock request type* specified in the lock request template establishes the machine action. The **lock request type** values are described in the following paragraphs.

•

- *Immediate request*- If the requested object location locks cannot be granted immediately, this option causes the *lock request not grantable* (hex 1A02) exception to be signaled. No object location locks are granted, and the lock request is canceled.
- *Synchronous request*- This option causes the thread requesting the locks to be placed in the wait state until all requested locks can be granted. If the locks cannot be granted in the time interval established

by the *wait time-out* field specified in the lock request template, the *object location lock wait time-out* (hex 3A05) exception is signaled to the requesting thread at the end of the interval. No locks are granted, and the lock request is canceled.

When:

- the thread requesting the locks is the only thread contained in the process at the time of the lock request, and
- the *lock request type* is *synchronous*, and
- the requested locks cannot be immediately granted,

the **access state modifications** field in the lock request template specifies whether the access state of the process access group is to be modified on entering and/or returning from the lock wait. The parameter has no effect if the process instruction wait access state control attribute specifies that no access state modification is allowed. If the process attribute value specifies that access state modification is allowed and the wait on event access state modification option specifies modify access state, the machine modifies the access state for the specified process access group.

If the thread requesting the locks belongs to a multi-threaded process, no access state modification is performed.

If the *lock request type* is *synchronous* and the invocation containing the Lock Object Location instruction is terminated, then the lock request is canceled.

Allocated object location locks are removed when the thread holding the locks terminates. If a thread terminates while waiting for an object location lock, the lock request is canceled.

The **asynchronous signals processing option** controls the action to be taken if an asynchronous signal is pending or received while in Lock Object Location wait. If an asynchronous signal that is not blocked or ignored is generated for the thread and the *asynchronous signals processing option* indicates *allow asynchronous signals processing during Lock Object Location wait*, the Lock Object Location wait will be terminated and an *asynchronous signal terminated MI wait* (hex 4C01) exception is signaled. Otherwise, when the *asynchronous signals processing option* indicates *do not allow asynchronous signals processing during Lock Object Location wait*, the thread remains in the wait until all requested locks can be granted or until the *wait time-out value for instruction* expires.

If any exception is identified during the instruction's execution, any locks already granted by the instruction are released, and the lock request is canceled.

For each object location lock, counts are kept by lock state and by thread. When a lock request is granted, the appropriate *lock count* of each lock state specified is incremented by 1.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1A Lock State

1A02 Lock Request Not Grantable

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C06 Machine Lock Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3A Wait Time-Out

3A05 Object Location Lock Wait Time-Out

4C Signals Management

4C01 Asynchronous Signal Terminated MI Wait

Lock Pointer-Based Mutex (LOCKMTX)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03D3	Mutex	Lock request template	Result

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Signed binary(4) variable scalar.

Bound program access

```
Built-in number for LOCKMTX is 157.
LOCKMTX (
         mutex                   : address
         lock_request_template   : address
) :   signed binary(4)   /* result */
```

Note: The term ″mutex″ in this instruction refers to a ″pointer-based mutex″.

Description: The *mutex*, whose address is contained in operand 1, is allocated exclusively to the issuing thread. Mutual exclusion is achieved between the thread with the allocated mutex lock and all other threads attempting to acquire the same mutex lock. If the mutex lock is successful, LOCKMTX returns with the issuing thread as its holder.

The *mutex* must have been previously created by the CRTMTX instruction or be a copy of a mutex that was previously created by the CRTMTX instruction prior to attempting to lock the mutex. See the CRTMTX instruction for additional information regarding mutex copies.

The *lock request template* whose address is passed in operand 2 is used to determine if and for how long the issuer will wait for the *mutex* to become available in the event the *mutex* is already locked. The lock options are used to place the issuer into the mutex wait state. If an invalid option is specified, an EINVAL error number is returned. If operand 2 in a bound program is a null pointer value, the default *lock request template* is used (the binary 0 value is the default action). Operand 2 in non-bound programs must be a pointer to a *lock request template*. The *pointer does not exist* (hex 2401) exception is signaled if a

null pointer value is used for operand 2 in a non-bound program. If the mutex lock can be immediately allocated to the issuer, then the *lock request template* is ignored.

Result is used to indicate the success or failure of the LOCKMTX instruction. If the *mutex* is locked by this instruction, then *result* is set to 0. If an error occurs, then the *result* is set to an error condition.

Mutexes can have either non-recursive or recursive behavior, which is specified at the time of creation. See the CRTMTX instruction for more information on mutex creation. A non-recursive mutex can only be locked once by a thread. Additional attempts to lock the same non-recursive mutex by a thread, will cause the EDEADLK error number to be returned. If a mutex was created as a recursive mutex then LOCKMTX can be used to recursively lock the mutex. The machine keeps track of the number of recursive locks for the mutex and requires that the thread use the Unlock Pointer-Based Mutex (UNLKMTX) instruction the same number of times to unlock the mutex before the mutex can be locked by a different thread. The maximum number of recursive locks is 32,767. The ERECURSE error number is returned if the maximum number of recursive locks is exceeded.

The *mutex* must be aligned on a 16-byte boundary.

The format of the *lock request template* follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Time-out option	Char(1)	
1	1	Lock options	Char(1)	
1	1		Reserved (must be 0)	Bit 0
1	1		Wait time-out format	Bit 1
			0 = Time-out value is specified in seconds/microseconds	
			1 = Time-out value is specified as a 64-bit binary value	
1	1		MPL (multiprogramming level) control during wait	Bit 2
			0 = Remain in the current MPL set	
			1 = Do not remain in the current MPL set	
1	1		Asynchronous signals processing option	Bit 3
			0 = Do not allow asynchronous signal processing during mutex wait	
			1 = Allow asynchronous signal processing during mutex wait	
1	1		Wait type	Bit 4
			0 = Normal wait	
			1 = Restricted wait	
1	1		Reserved (binary 0)	Bits 5-7
2	2	Reserved (binary 0)	Char(6)	
8	8	Wait time-out value	Char(8)	
8	8		Seconds	Bin(4)
12	C		Microseconds	Bin(4)
16	10	—- End —-		

The **time-out option** describes the action taken when LOCKMTX cannot allocate the *mutex* to the issuer immediately.

Hex 00 =	Wait indefinitely for the mutex.
Hex 01 =	Wait for the mutex for the specified amount of time. If the mutex still cannot be obtained, the EAGAIN error number is returned.
Hex 02 =	Return immediately with an EBUSY error number.

All other values for *time-out option* are reserved.

The **lock options** describes the action taken when LOCKMTX causes the issuer to be put into a mutex wait state.

The **wait time-out format** option determines the format of the *wait time-out value*. If the *wait time-out format* is specified as 0, then the **wait time-out value** consists of two 4-byte fields, specifying the number of seconds and the number of microseconds to wait for the *mutex* if it cannot be locked immediately. If the *wait time-out format* is specified as 1, then the *wait time-out value* consists of a single 8-byte field. See "Standard Time Format" on page 1272 for additional information on the format of the *wait time-out value* if *wait time-out format* is specified as 1.

The **MPL control during wait** option controls whether the thread is removed from the current MPL (multiprogramming level) set or remains in the current MPL set when the thread enters a mutex wait. When entering a mutex wait state, the thread will normally remain in the current MPL set for an implementation-defined period which will not exceed 2 seconds. If the mutex wait has not been satisfied by the end of this period, the thread is automatically removed from the current MPL set. The automatic removal does not change or affect the total wait time specified on the LOCKMTX instruction. If the *MPL control during wait* option specifies do not remain in current MPL set (value of 1), then the thread will leave the MPL set immediately.

The **asynchronous signals processing option** controls the action to be taken if an asynchronous signal is pending or received while in a mutex wait. If an asynchronous signal that is not blocked or ignored is generated for the thread and the *asynchronous signals processing option* indicates *allow asynchronous signal processing during mutex wait*, the mutex wait is terminated and the result set to EINTR. Otherwise, when the *asynchronous signals processing option* indicates *do not allow asynchronous signal processing during mutex wait*, the thread remains in the wait until the *mutex* is allocated exclusively to the thread or until the *wait time-out value* expires.

The **wait type** option is used by kernel-mode programs or procedures to specify what type of wait to perform. The *wait type* field is ignored when the thread execution mode is not kernel-mode.

The **wait time-out value** establishes the maximum amount of time that the issuer waits for the requested *mutex* when the mutex lock cannot be immediately obtained and the *time-out option* is set to *wait for a specified period of time* (hex 01). The format of this field differs depending on which value is specified by the *wait time-out format*. If the *wait time-out format* is 0, then the *wait time-out value* consists of two 4-byte values, specifying the number of **seconds** and the number of **microseconds** to wait for the *mutex*. If the *wait time-out format* is 1, then the *wait time-out value* is a 64-bit unsigned binary value, with the bits being numbered from 0 to 63, and bit 48 is equal to 8 microseconds. See "Standard Time Format" on page 1272 for additional information on the format of the *wait time-out value* if *wait time-out format* is specified as 1.

Note: Regardless of the format used for the *wait time-out value*, the timer is architected to be updated once every 8 microseconds. Increased granularity of the time-out period cannot be assumed by specifying *wait time-out format* as 0 rather than as 1.

The maximum time-out value allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum time-out value causes the maximum time-out value to be used. If the *wait*

time-out value is binary 0, then the process default wait time-out value is used. This field is only applicable if the *time-out option* is set to hex 01. If the *time-out option* is not set to hex 01, this field is ignored.

If a mutex is destroyed while another thread has a pending request to lock the *mutex*, an EDESTROYED error number is returned to the waiting thread. The ETYPE error number is returned if the *mutex* operand references a synchronization object that is not a pointer-based mutex.

The *keep valid option* on the Create Mutex (CRTMTX) instruction specifies whether the *mutex* is to remain valid when a thread is terminated while holding the lock on the *mutex*. The *keep valid option* must be set to hex 01 if a mutex should be remain valid after thread termination. The mutex is considered to be in a pending state when it is kept valid after thread termination. The next thread to lock a pending mutex will revalidate the mutex, but will receive an EUNKNOWN error number to indicate that the resource protected by this mutex may need special handling. Appropriate action is left up to the MI user's discretion. Specifying hex 00 for the *keep valid option* will result in the locked mutex being destroyed during thread termination. Threads waiting on the destroyed mutex will receive an EOWNERTERM error number.

If the space where a mutex resides is destroyed or corrupted, the system may detect this condition and attempt to clean up the mutex resource. This cleanup action will cause any thread waiting on the mutex to awaken with a return value of EINVAL.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Error conditions

The *result* is set to one of the following:

EAGAIN 3406 - Operation would have caused the process to be suspended.

EBUSY 3029 - Resource busy.

ECANCEL 3456 - Operation canceled.

EDEADLK 3459 - Resource deadlock avoided.

EDESTROYED 3463 - The synchronization object was destroyed, or the object no longer exists.

EINTR 3407 - Interrupted function call.

EINVAL 3021 - The value specified for the argument is not correct.

EOWNERTERM 3462 - The synchronization object no longer exists because the owner is no longer running.

ERECURSE 3419 - Recursive attempt rejected.

Recursion limit exceeded for a recursive mutex.

ETERM 3464 - Operation terminated.

ETYPE 3493 - Object type mismatch.

A synchronization object at this address is not a pointer-based mutex.

EUNKNOWN 3474 - Unknown system state.

Owner of mutex was previously terminated leaving the mutex in a pending state. EUNKNOWN is returned to the first locker following this scenario. See CRTMTX for additional information.

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Lock Space Location (LOCKSL)

Op Code (Hex)	Operand 1	Operand 2
03F6	Space location or lock request template	Lock request

Operand 1: Space pointer data object.

Operand 2: Character(1) scalar or null.

Bound program access

```
Built-in number for LOCKSL is 47.
LOCKSL (
        space_location   : address of space pointer(16)
        lock_request     : address OR
                           null operand
)
```

Description: When operand 2 is not null, a thread scoped lock specified by operand 2 is requested by the current thread for the *space location* (operand 1). When operand 2 is null, thread scoped space location locks identified in the *lock request template* (operand 1) are requested by the current thread.

Locking a space location does not prevent any byte operation from referencing that location, nor does it prevent the space from being extended, truncated, or destroyed. Space location locks follow the normal locking rules with respect to conflicts and waits but are strictly symbolic in nature.

A space pointer machine object cannot be specified for operand 1. If operand 1 points to teraspace an *unsupported space use* (hex 0607) exception is signaled. The LOCKTSL instruction can however be used with teraspace locations.

The following is the format of operand 2 when not null:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Lock request	Char(1)	
0	0		Lock state selection (1 = lock requested, 0 = lock not requested) Only one state may be requested.	Bits 0-4
0	0		LSRD lock	Bit 0
0	0		LSRO lock	Bit 1
0	0		LSUP lock	Bit 2
0	0		LEAR lock	Bit 3
0	0		LENR lock	Bit 4
0	0		Reserved (binary 0)	Bits 5-7
1	1	—- End —-		

For this format, if the requested lock cannot be immediately granted, the thread will enter a synchronous wait for the lock for a period of up to the interval specified by the process default time-out value. If the wait exceeds this time limit, a *space location lock wait time-out* (hex 3A04) exception is signaled, and the requested lock is not granted.

During the wait, the process access state may be modified. This can occur if the process' *instruction wait access state* control attribute is set to *allow access state modification*, and if the process contains a single thread at the time of the lock request. Process access states are not modified for multi-threaded processes.

When operand 2 is null, the *lock request template* identified by operand 1 must be aligned on a 16-byte boundary. The format of operand 1 is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Number of space location lock requests in the template	Bin(4)
4	4	Offset to lock state selection values	Bin(2)
6	6	Wait time-out value for instruction	Char(8)
14	E	Lock request options	Char(3)
14	E		Reserved (binary 0)
14	E		Lock request type **0 =** Immediate request-If all locks cannot be immediately granted, signal exception. **1 =** Synchronous request-Wait until all locks can be granted.
14	E		Access state modifications
14	E		When the process is entering lock wait for synchronous **0 =** Access state should not be modified. **1 =** Access state should be modified.
14	E		When the process is leaving lock wait: **0 =** Access state should not be modified. **1 =** Access state should be modified.
14	E		Reserved (binary 0)
14	E		Time-out option

Offset Dec	Hex	Field Name	Data Type and Length
			0 = Wait for specified time, then signal time-out exception.
			1 = Wait indefinitely.
14	E		Reserved (binary 0)
14	E		Modify thread event mask option
			0 = Do not modify thread event mask
			1 = Modify thread event mask
14	E		Asynchronous signals processing option
			0 = Do not allow asynchronous signal processing during Lock Space Location wait.
			1 = Allow asynchronous signal processing during Lock Space Location wait.
14	E		Reserved (binary 0)
17	11	Modify thread event mask control	Char(4)
17	11		New thread event mask
19	13		Previous thread event mask
21	15	Reserved (binary 0)	Char(11)
32	20	Space location(s) to be locked	[*] Space pointer
		This should be repeated as specified by *number of space location lock requests in template* above.	
*	*	—- End —-	

The *lock state selection* is located by adding the *offset to lock state selection values* above to operand 1.

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Lock state selection	[*] Char(1)	
		(repeated for each pointer in the template)		
0	0		Requested lock state	Bits 0-4
			(1 = lock requested, 0 = lock not requested) Only one state may be requested per entry.	
0	0		LSRD lock	Bit 0
0	0		LSRO lock	Bit 1
0	0		LSUP lock	Bit 2
0	0		LEAR lock	Bit 3
0	0		LENR lock	Bit 4
0	0		Reserved (binary 0)	Bits 5-6
0	0		Entry active indicator	Bit 7
			0 = Entry not active- This entry is not used.	
			1 = Entry active- Obtain this lock.	
*	*	—- End —-		

Lock Allocation Procedure: A single Lock Space Location instruction can request the allocation of one or more lock states on one or more space locations. Space location locks are granted sequentially until all the locks requested are granted.

The **wait time-out** field establishes the maximum amount of time that a thread competes for the requested set of locks when the *lock request type* is *synchronous*. See "Standard Time Format" on page 1272 for additional information on the format of the *wait time-out*. The maximum wait time-out interval allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum wait time-out causes the maximum wait time-out to be used. If the *wait time-out* field is specified with a value of binary 0, then the value associated with the default wait time-out parameter in the process definition template establishes the time interval.

When a requested lock state cannot be immediately granted, any locks already granted by this Lock Space Location instruction are released, and the *lock request type* specified in the lock request template establishes the machine action. The **lock request type** values are described in the following paragraphs.

-
- *Immediate request*- If the requested space location locks cannot be granted immediately, this option causes the *lock request not grantable* (hex 1A02) exception to be signaled. No space location locks are granted, and the lock request is canceled.
- *Synchronous request*- This option causes the thread requesting the locks to be placed in the wait state until all requested locks can be granted. If the locks cannot be granted in the time interval established by the *wait time-out* field specified in the lock request template, the *space location lock wait time-out* (hex 3A04) exception is signaled to the requesting thread at the end of the interval. No locks are granted, and the lock request is canceled.

When:

- the thread requesting the locks is the only thread in the process at the time of the lock request, and
- the *lock request type* is *synchronous*, and
- the requested locks cannot be immediately granted,

the **access state modifications** field in the lock request template specifies whether the access state of the process access group is to be modified on entering and/or returning from the lock wait. The parameter has no effect if the process instruction wait access state control attribute specifies that no access state modification is allowed. If the process attribute value specifies that access state modification is allowed and the wait on event access state modification option specifies modify access state, the machine modifies the access state for the specified process access group.

If the thread requesting the locks belongs to a multi-threaded process, no access state modification is performed.

If the *lock request type* is *synchronous* and the invocation containing the Lock Space Location instruction is terminated, then the lock request is canceled.

Allocated space location locks are removed when the thread holding the locks terminates. If a thread terminates while waiting for a space location lock, the lock request is canceled.

The **modify thread event mask option** controls the state of the event mask in the thread executing this instruction. When the thread event mask is in the *masked* state, the machine does not schedule signaled event monitors in the thread. The event monitors continue to be signaled by the machine or other threads. When the thread event mask is modified to the *unmasked* state, event handlers are scheduled to handle those events that occurred while the thread was masked and those events occurring while in the unmasked state.

If the system security level machine attribute is hex 40 or greater and the thread is running in user state, then the *modify thread event mask option* is not allowed and a *template value invalid* (hex 3801) exception is signalled.

When the *modify thread event mask* is set to 1, the **previous thread event mask** will be returned and the **new thread event mask** will take effect only when the space location lock(s) have been successfully granted. If the space location lock request is not successful, the *previous thread event mask* value is not returned, nor does the *new thread event mask* take effect.

The thread event mask values are validity checked only when the modify thread event mask is set to 1, and ignored otherwise. Valid masking values are:

0	Masked
256	Unmasked

Other values are reserved and must not be specified, otherwise a *template value invalid* (hex 3801) exception is signaled.

The **asynchronous signals processing option** controls the action to be taken if an asynchronous signal is pending or received while in Lock Space Location wait. If an asynchronous signal that is not blocked or ignored is generated for the thread and the *asynchronous signals processing option* indicates *allow asynchronous signals processing during Lock Space Location wait*, the Lock Space Location wait will be terminated and an *asynchronous signal terminated MI wait* (hex 4C01) exception is signaled. Otherwise, when the *asynchronous signals processing option* indicates *do not allow asynchronous signals processing during Lock Space Location wait*, the thread remains in the wait until all requested locks can be granted or until the *wait time-out value for instruction* expires.

If any exception is identified during the instruction's execution, any locks already granted by the instruction are released, and the lock request is canceled.

For each space location lock, counts are kept by lock state and by thread. When a lock request is granted, the appropriate *lock count* of each lock state specified is incremented by 1.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1A Lock State

- 1A02 Lock Request Not Grantable

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded
- 1C06 Machine Lock Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

- 3201 Scalar Type Invalid
- 3203 Scalar Value Invalid

36 Space Management

- 3601 Space Extension/Truncation

38 Template Specification

- 3801 Template Value Invalid

3A Wait Time-Out

3A04 Space Location Lock Wait Time-Out

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

4C Signals Management

4C01 Asynchronous Signal Terminated MI Wait

Lock Teraspace Storage Location (LOCKTSL)

Op Code (Hex)	Operand 1
0315	Lock request template

Operand 1: Space pointer data object.

Bound program access

```
Built-in number for LOCKTSL is 621.
LOCKTSL (
         lock_request_template   : address of space pointer(16)
)
```

Description: The locks identified in the *lock request template* (operand 1) are requested by the current thread. By default, these locks are obtained thread scoped. Process and Transaction Control Structure scoped locks are also supported. The LOCKTSL instruction can be used to lock teraspace storage locations symbolically or as resolved single-level store addresses, as well as any location which can be locked by the LOCKSL instruction. A maximum of 4093 locations can be locked with one LOCKTSL instruction.

If the *type of teraspace storage location lock* field in the *lock request template* is set to binary 0, then all teraspace storage locations will be locked symbolically. This implies that only the thread which is performing the lock operation and any other threads in the same process will ever conflict on teraspace storage location locks.

If the *type of teraspace storage location lock* field in the *lock request template* is set to binary 1, then the resolved single-level store location mapped to the teraspace storage location will be locked. This requires that each teraspace storage location be either previously allocated or mapped. If they are not, an *unsupported space use* (hex 0607) exception is signaled and no locations are locked.

If multiple threads, either in the same process or different processes, resolve their teraspace storage addresses to a common single-level store location, and the *type of teraspace storage location lock* field was set to binary 1 when the lock requests were made by the threads, then the locks will conflict according to the normal locking rules, even though the individual teraspace storage addresses are apparently different.

If the location to be locked is not a teraspace storage location, the *type of teraspace storage location lock* field is ignored.

Locking a teraspace storage location does not prevent any byte operation from referencing that location, nor does it prevent the space from being extended, truncated, or destroyed. Teraspace storage location locks follow the normal locking rules with respect to conflicts and waits.

The *lock request template* identified by operand 1 must be aligned on a 16-byte boundary or an *boundary alignment* (hex 0602) exception is signaled. The format of operand 1 is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Number of lock requests in the template (max 4093)	UBin(4)
4	4	Offset to lock state selection values	UBin(2)
6	6	Wait time-out value for instruction	Char(8)
14	E	Lock request options	Char(3)
14	E		Reserved (binary 0)
14	E		Lock request type **0 =** Immediate request. **1 =** Synchronous request.
14	E		Access state modifications
14	E		When the process is entering lock wait for synchronous **0 =** Access state should not be modified. **1 =** Access state should be modified.
14	E		When the process is leaving lock wait for synchronous r **0 =** Access state should not be modified. **1 =** Access state should be modified.
14	E		Type of teraspace storage location lock **0 =** Lock the teraspace storage location symbolically. **1 =** Lock the resolved single-level store location mapped to the teraspace storage location.
14	E		Reserved (binary 0)
14	E		Time-out option **0 =** Wait for specified time, then signal time-out exception. **1 =** Wait indefinitely.
14	E		Reserved (binary 0)
14	E		Lock scope **0 =** Lock is scoped to the current thread **1 =** Lock is scoped to the *lock scope object type*
14	E		Lock scope object type **0 =** Process containing the current thread **1 =** Transaction control structure attached to the current thread.
14	E		Reserved (binary 0)
14	E		Modify thread event mask option **0 =** Do not modify thread event mask **1 =** Modify thread event mask
14	E		Asynchronous signals processing option

Offset Dec	Hex	Field Name	Data Type and Length
			0 = Do not allow asynchronous signal processing during Lock Teraspace Storage Location wait.
			1 = Allow asynchronous signal processing during Lock Teraspace Storage Location wait.
14	E		Reserved (binary 0)
17	11	Modify thread event mask control	Char(4)
17	11		New thread event mask
19	13		Previous thread event mask
21	15	Reserved (binary 0)	Char(11)
32	20	Location(s) to be locked	[*] Space pointer
		(This should be repeated as specified by *number of lock requests in the template* above.)	
*	*	—- End —-	

The *lock state selection* is located by adding the *offset to lock state selection values* above to operand 1.

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Lock state selection	[*] Char(1)	
		(repeated for each pointer in the template)		
0	0		Requested lock state	Bits 0-4
			(1 = lock requested, 0 = lock not requested) Only one state may be requested per entry.	
0	0		LSRD lock	Bit 0
0	0		LSRO lock	Bit 1
0	0		LSUP lock	Bit 2
0	0		LEAR lock	Bit 3
0	0		LENR lock	Bit 4
0	0		Reserved (binary 0)	Bits 5-6
0	0		Entry active indicator	Bit 7
			0 = Entry not active- This entry is to be ignored.	
			1 = Entry active- Obtain this lock.	
*	*	—- End —-		

Lock Allocation Procedure: A single Lock Teraspace Storage Location instruction can request the allocation of one or more lock states on one or more storage locations. The locks are granted sequentially until all the locks requested are granted. However, all teraspace storage locations are treated the same as specified by the *type of teraspace storage location lock* field in the template.

The **type of teraspace storage location lock** field determines if teraspace storage locations are to be locked symbolically or if the single-level store location mapped to the teraspace storage location will be locked. If this field is set to binary 1, then all teraspace storage locations specified in the template will be resolved to their single-level store location before being locked. This allows locks on shared memory mapped locations to conflict even though the teraspace storage locations are different. The teraspace storage location is required to have been previously allocated or mapped by each thread performing the lock operation. If the teraspace storage location has not been allocated or mapped, an *unsupported space use* (hex 0607) exception is signaled.

If the *type of teraspace storage location lock* field is set to binary 0, then all teraspace storage locations specified in the template will be treated as symbolic locations. Therefore, if two different teraspace storage locations map to the same single-level store location, and the field is set to binary 0, then the locks will never conflict, even within a single process.

Note that when locks are materialized using Materialize Process Locks, (MATPRLK), if the lock was acquired with the *type of teraspace storage location lock* field set to binary 1, then a null pointer value will be returned. In addition, if the process whose locks are being materialized is not the current process, then a null pointer value is returned for any lock on a teraspace storage location. If the current process is materializing its own locks, then a space pointer to the teraspace is generated when the lock was acquired with the *type of teraspace storage location lock* field set to binary 0.

The **wait time-out value for instruction** field establishes the maximum amount of time that a thread competes for the requested set of locks when the *lock request type* is *synchronous*. See "Standard Time Format" on page 1272 for additional information on the format of the *wait time-out value for instruction* field. The maximum wait time-out interval allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum wait time-out interval causes the maximum wait time-out interval to be used. If the *wait time-out value for instruction* field is set to a value of binary 0, then the value associated with the default wait time-out parameter in the process definition template is used.

When a requested lock state cannot be immediately granted, any locks already granted by this Lock Teraspace Storage Location instruction are released and the *lock request type* specified in the lock request template establishes the machine action. The **lock request type** values are described in the following paragraphs.

-
- *Immediate request*- If the requested locks cannot be granted immediately, this option causes the *lock request not grantable* (hex 1A02) exception to be signaled. No location locks are granted, and the lock request is canceled.
- *Synchronous request*- This option causes the thread requesting the locks to be placed in the wait state until all requested locks can be granted. If the locks cannot be granted in the time interval established by the *wait time-out value for instruction* field specified in the lock request template, a *space location lock wait time-out* (hex 3A04) exception is signaled to the requesting thread at the end of the interval. No locks are granted, and the lock request is canceled.

When:

- the thread requesting the locks is the only thread in the process at the time of the lock request, and
- the *lock request type* is *synchronous*, and
- the requested locks cannot be immediately granted,

the **access state modifications** field in the lock request template specifies whether the access state of the process access group is to be modified on entering and/or returning from the lock wait. The parameter has no effect if the process instruction wait access state control attribute specifies that no access state modification is allowed. If the process attribute value specifies that access state modification is allowed and the wait on event access state modification option specifies modify access state, the machine modifies the access state for the specified process access group.

If the *lock request type* is *synchronous* and the invocation containing the Lock Teraspace Storage Location instruction is terminated, then the lock request is canceled.

Allocated space location locks are removed when the thread holding the locks terminates. If a thread terminates while waiting for a teraspace storage location lock, the lock request is canceled.

The **lock scope** field and the **lock scope object type** field determine which scope all specified lock requests will be allocated to, either a thread, process or transaction control structure:

•
- When *lock scope* has a value of *lock is scoped to the lock scope object type* and *lock scope object type* has a value of *process containing the current thread*, the lock scope will be the process containing the current thread.
- When *lock scope* has a value of *lock is scoped to the lock scope object type* and *lock scope object type* has a value of *transaction control structure attached to the current thread*, the lock scope will be the transaction control structure that is attached to the current thread. If the current thread does not have a transaction control structure attached, then the lock scope will be the process containing the current thread.
- When *lock scope* has a value of *lock is scoped to the current thread*, the lock scope will be to the current thread.

Locks scoped to a thread with a *lock space object type* value of *process containing the current thread* can never conflict with a lock scoped to its containing process, but may conflict with a lock scoped to a different process, a transaction control structure, or any other thread (depending on the lock states involved). Locks scoped to a thread with a *lock space object type* value of *transaction control structure attached to the current thread* can never conflict with a lock scoped to the transaction control structure, but may conflict with a lock scoped to a different transaction control structure, a process, or any other thread (depending on the lock states involved).

If *lock scope object type* has a value of *transaction control structure attached to the current thread* and the transaction control structure state does not allow objects to be locked on behalf of the transaction control structure, a *object not eligible for operation* (hex 2204) exception is signaled.

Allocated process scope locks and any thread scoped locks, allocated by the initial thread of the process, are released when the initial thread terminates. Allocated thread scope locks are released when the thread terminates. If a thread requested a process scope lock, the process will continue to hold that lock after termination of the requesting thread. If a thread requested a transaction control structure scope lock, the transaction control structure will continue to hold that lock after the termination of the requesting thread.

The **modify thread event mask option** controls the state of the event mask in the thread executing this instruction. When the thread event mask is in the *masked* state, the machine does not schedule signaled event monitors in the thread. The event monitors continue to be signaled by the machine or other threads. When the thread event mask is modified to the *unmasked* state, event handlers are scheduled to handle those events that occurred while the thread was masked and those events occurring while in the unmasked state.

If the system security level machine attribute is hex 40 or greater and the thread is running in user state, then the *modify thread event mask option* is not allowed and a *template value invalid* (hex 3801) exception is signalled.

When the *modify thread event mask* is set to 1, the **previous thread event mask** will be returned and the **new thread event mask** will take effect only when lock(s) have been successfully granted. If the lock request is not successful, the *previous thread event mask* value is not returned, nor does the *new thread event mask* take effect.

The thread event mask values are validity checked only when the modify thread event mask is set to 1, and ignored otherwise. Valid masking values are:

0	Masked
256	Unmasked

Other values are reserved and must not be specified, otherwise a *template value invalid* (hex 3801) exception is signaled.

The **asynchronous signals processing option** controls the action to be taken if an asynchronous signal is pending or received while in lock wait. If an asynchronous signal that is not blocked or ignored is generated for the thread and the *asynchronous signals processing option* indicates *allow asynchronous signals processing during Lock Teraspace Storage Location wait*, the Lock Teraspace Storage Location wait will be terminated and an *asynchronous signal terminated MI wait* (hex 4C01) exception is signaled. Otherwise, when the *asynchronous signals processing option* indicates *do not allow asynchronous signals processing during Lock Teraspace Storage Location wait*, the thread remains in the wait until all requested locks can be granted or until the *wait time-out value for instruction* expires.

If any exception is identified during the instruction's execution, any locks already granted by the instruction are released, and the lock request is canceled.

For each lock, counts are kept by lock state and by thread. When a lock request is granted, the appropriate *lock count* of each lock state specified is incremented by 1.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1A Lock State

1A02 Lock Request Not Grantable

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C06 Machine Lock Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2204 Object Not Eligible for Operation

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3A Wait Time-Out

3A04 Space Location Lock Wait Time-Out

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

4C Signals Management

4C01 Asynchronous Signal Terminated MI Wait

Logarithm Base E (Natural Logarithm) (LN)

Bound program access
Built-in number for LN is 406. LN (source : floating point(8) value) : floating point(8) value which is the natural logarithm of the source value

Description: The natural logarithm of the *source* operand value is computed and the result is returned.

The result is in the range:

```
-infinity <= LN(source) <= +infinity
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Materialize Access Group Attributes (MATAGAT)

Op Code (Hex)	Operand 1	Operand 2
03A2	Receiver	Access group

Operand 1: Space pointer.

Operand 2: System pointer.

Bound program access
Built-in number for MATAGAT is 68. MATAGAT (receiver : address access_group : address of system pointer)

Description: The attributes of the *access group* and the identification of objects currently contained in the access group are materialized into the receiving object specified by operand 1.

Objects requested to be in the access group may:

-
- exist entirely in the access group,
- exist partially in the access group and partially outside the access group,
- or exist entirely outside the access group.

The machine may also use the access group for enabling programs to run within a process. In this case, the Process Control Space (PCS) object is considered to exist partially in the access group, even if the access group membership was not requested when the PCS was created.

Only objects which exist wholly or partially in the *access group* will be materialized.

The materialization must be aligned on a 16-byte boundary. The format is:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Object identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Object creation options	Char(4)	
40	28		Existence attributes	Bit 0
			0 = Temporary	
			1 = Reserved	
40	28		Space attribute	Bit 1
			0 = Fixed-length	
			1 = Variable-length	
40	28		Context	Bit 2
			0 = Addressability not in context	
			1 = Addressability in context	
40	28		Reserved (binary 0)	Bits 3-12
40	28		Initialize space	Bit 13
40	28		Reserved (binary 0)	Bits 14-31
44	2C	Reserved (binary 0)	Char(4)	
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	

Offset Dec	Offset Hex	Field Name	Data Type and Length		
53	35	Performance class	Char(4)		
53	35		Space alignment		Bit 0
			0 =	The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If no space is specified for the object, this value must be specified for the performance class.	
			1 =	The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space.	
53	35		Reserved (binary 0)		Bits 1-4
53	35		Default main storage pool		Bit 5
			0 =	Process main storage pool is used for this object.	
			1 =	Machine default main storage pool is used for this object.	
53	35		Reserved (binary 0)		Bit 6
53	35		Block transfer on implicit access state modification		Bit 7
			0 =	Transfer the minimum storage transfer size for this object.	
			1 =	Transfer the machine default storage transfer size for this object.	
53	35		Reserved (binary 0)		Bits 8-31
57	39	Reserved (binary 0)	Char(7)		
64	40	Context	System pointer		
80	50	Reserved (binary 0)	Char(16)		
96	60	Access group size	UBin(4)		
100	64	Available space in the access group	UBin(4)		
104	68	Number of objects in the access group	UBin(4)		
108	6C	Reserved (binary 0)	Char(4)		
112	70	Access group object system pointer	[*] System pointer		
		(repeated for each object currently contained in the access group)			
*	*	—- End —-			

The *receiver* space contains the access group's attributes, the current status of the access group, and a system pointer to each object assigned to the access group.

The **number of bytes provided** indicates the size of the materialization template. The **number of bytes available** is set by the instruction to indicate the actual number of bytes available to be returned. In no case does the instruction return more bytes of information than *number of bytes provided*.

The **access group size** represents the total amount of space that has been allocated to the access group.

The **available space in the access group** represents the amount of space that is available in the access group for additional objects.

The **number of objects in the access group** is a count of the number of objects that are currently contained in the access group. This value is also the number of times that the *access group object system pointer* below is repeated.

There is one **access group object system pointer** for each object currently assigned to the access group. The authorization field within each system pointer is not set.

Warning: Temporary Level 3 Header

Authorization Required

•
- Retrieve
  –
  – Operand 2
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•
- Materialize
  –
  – Operand 2
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Activation Attributes (MATACTAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
MATACTAT2 0233	Receiver	Activation mark	Attribute selection
MATACTAT 0213	Receiver	Activation mark	Attribute selection

Operand 1: Space pointer.

Operand 2 for MATACTAT2: Char(8) scalar.

Operand 2 for MATACTAT: Unsigned binary(4) scalar.

Operand 3: Character(1) scalar.

Bound program access

```
Built-in number for MATACTAT2 is 659.
MATACTAT2 (
          receiver              : address
          activation_mark       : address of unsigned binary(8)
          attribute_selection   : address
)

OR
Built-in number for MATACTAT is 121.
MATACTAT (
          receiver              : address
          activation_mark       : address of unsigned binary(4)
          attribute_selection   : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note

It is recommended that you use the MATACTAT2 instruction which supports 8-byte activation marks and that you use the 8-byte activation and activation group marks at the end of the Basic Activation Attributes template. 4-byte marks can wrap and produce unexpected results.

Description: This instruction will materialize the information selected by operand 3 for the program activation specified by operand 2 and return the information in the template supplied by operand 1.

The operand 3 selection operand is provided to deal with the variable-length nature of some of the returned information. All ″length-of-list″ type information can be gathered by selecting the first option described below.

Operand 3 can have the following values:

•
- Hex 00 — basic activation attributes
- Hex 01 — static storage frame list
- Hex 02 — dependent activation mark list

  MATACTAT2 returns a list of 8-byte activation marks. MATACTAT returns a list of 4-byte activation marks.

Any value for operand 3 other than those listed will cause a *scalar value invalid* (hex 3203) exception.

Operand 2 is different for the MATACTAT and MATACTAT2 instructions.

Operand 2 for MATACTAT2: Operand 2 supplies the 8-byte activation mark of the activation for which information is to be returned.

Operand 2 for MATACTAT: Operand 2 supplies the 4-byte activation mark of the activation for which information is to be returned.

The activation mark uniquely identifies an activation within a process. A value of zero is interpreted to be a request for information about the activation of the invoking program.

The materialization template identified by operand 1 must be 16-byte aligned in the space. This materialization template has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size specification	Char(16)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8		Reserved (binary 0)	Char(8)
16	10	Returned information	Char(*)	
*	*	—- End —-		

The **number of bytes provided** indicates the size of the materialization template. This includes the length of the template header (16 bytes) plus the number of bytes provided for *returned information*. If a value of 8 is specified, then no data will actually be materialized and the number of bytes required to materialize the requested data will be returned in *number of bytes available*. Note that a value greater than 8, but less than 16 will result in no data being materialized, since bytes 9-16 are reserved. If the number of bytes provided is less than 8, then a *materialization length invalid* (hex 3803) exception is signaled.

The **number of bytes available** is set by the instruction to indicate the actual number of bytes available to be returned. The *number of bytes available* also includes the length of the template header (16 bytes). In no case does the instruction return more bytes of information than those available.

The format of **returned information** is described in the following paragraphs.

Basic Activation Attributes: The following information is returned when operand 3 is hex 00.

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Program	System pointer
32	20	Activation mark	UBin(4)
36	24	Activation group mark	UBin(4)
40	28	Invocation count	UBin(4)
44	2C	Static frame count	UBin(4)
48	30	Program type	Char(1)

Hex 00 =
Non-bound program

Hex 01 =
Bound program, bound service program, or Java(TM) program

Hex 02-FF =
Reserved

Offset Dec	Hex	Field Name	Data Type and Length	
49	31	Activation attributes	Char(1)	
49	31		Activation status	Bit 0
			0 = Inactive	
			1 = Active	
49	31		Reserved (binary 0)	Bits 1-7
50	32	Target activation group	Char(1)	
		0 = Default activation group		
		1 = Caller's activation group		
		2 = Named activation group		
		3 = Unnamed activation group		
		4 = Named shared activation group		
		5 = Unnamed shared activation group		
		6-255 Reserved		
51	33	Reserved (binary 0)	Char(1)	
52	34	Dependent activation count	UBin(4)	
56	38	Activation mark	UBin(8)	
		For Non-Bound programs, the following datatype should be used:		
56	38		Activation mark (Non-Bound program)	Char(8)
64	40	Activation group mark	UBin(8)	
		For Non-Bound programs, the following datatype should be used:		
64	40		Activation group mark (Non-Bound program)	Char(8)
72	48	—- End —-		

A description of the fields follows.

Program
This is a pointer to the program. The system pointer returned does not contain authority. Within a process, a program may have more than one activation.

Activation mark
The *activation mark* identifies the activation within the process. This field provides the actual activation mark when the special zero value was supplied for operand 2. Otherwise, this field has the same value as operand 2. The value returned in the 4-byte activation mark may have wrapped.

Activation group mark
This identifies the activation group which contains the activation. The value returned in the 4-byte activation group mark may have wrapped.

Invocation count
This is a count of the number of program invocations which currently exist for this activation of the program. The count includes all threads in the process which owns the identified activation. Recall that a program invocation results from a *program call* operation like Call Program.

Static frame count
This is the number of static storage frames allocated for this activation.

Program type
The type of the program. A program is either a *non-bound program*, *bound program*, *bound service program* or *Java program*.

Activation status
The activation status identifies whether the program is active.

Target activation group
This is the *target activation group* attribute of the program object associated with this activation.

Dependent activation count
This is the number of dependent program activations directly bound to the program identified by the activation mark in operand 2.

Static Storage Frame List: The following information is returned when operand 3 is hex 01. This is a list of static storage frame descriptors. The *static frame count* (available in the basic activation attributes template) indicates how many entries must be accommodated by the template. The static storage frame list can be materialized only if the source activation group is permitted access to the target activation group as determined by the activation group access protection mechanism. If access is not permitted, then an *activation group access violation* (hex 2C12) exception is signaled.

The format of the list is:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Static storage frame list entry (repeated *static frame count* times)	[*] Char(32)	
16	10		Static frame base This is a pointer to the first byte of the static frame.	Space pointer
32	20		Static frame size This is the size, in machine dependent units (currently bytes), of the static frame.	UBin(4)
36	24		Reserved	Char(12)
*	*	—- End —-		

Dependent Activation Mark List: The following information is returned when operand 3 is hex 02. This is a list of activation marks of all the dependent programs directly bound to the program specified in operand 2. The *dependent activation count* (available in the basic activation attributes template) indicates how many entries must be accommodated by the template. The format of the list is different for the MATACTAT and MATACTAT2 instructions.

Format of the dependent activation mark list for MATACTAT2:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Activation mark (repeated *dependent activation count* times) This is the activation mark of a dependent program activation. For Non-Bound programs, the following datatype should be used:	[*] UBin(8)	
16	10		Activation mark (Non-Bound program)	Char(8)
*	*	—- End —-		

Format of the dependent activation mark list for MATACTAT:

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Activation mark	[*] UBin(4)

Offset Dec	Hex	Field Name	Data Type and Length
		(repeated *dependent activation count* times) This is the activation mark of a dependent program activation. The value returned in this field may have wrapped.	
*	*	—- End —-	

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C09 Auxiliary Storage Pool Number Invalid

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2C Program Execution

2C11 Process Object Access Invalid

2C12 Activation Group Access Violation

2C16 Program Activation Not Found

32 Scalar Specification

3203 Scalar Value Invalid

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Activation Export (MATACTEX)

Bound program access

```
Built-in number for MATACTEX2 is 660.
MATACTEX2 (
          activation_mark   : unsigned binary(8)
          ident_type        : unsigned binary(4)
          number            : unsigned binary(4)
          name              : address
          pointer           : address of procedure pointer(16) OR
                              address of space pointer(16)
          export_type       : address of unsigned binary(4)
)
OR

Built-in number for MATACTEX is 460.
MATACTEX (
          activation_mark   : unsigned binary(4)
          ident_type        : unsigned binary(4)
          number            : unsigned binary(4)
          name              : address
          pointer           : address of procedure pointer(16) OR
                              address of space pointer(16)
          export_type       : address of unsigned binary(4)
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note

It is recommended that you use the MATACTEX2 instruction which supports 8-byte activation marks. 4-byte marks can wrap and produce unexpected results.

Description: This instruction returns the address of an export identified by name or export identifier from a specified program activation. The type of the export item, either data or procedure, is also returned. If the item is not found, an indicator is returned via the *export type* operand; no exception is signaled in this case.

Operands are as follows,

•

- *activation mark* (input) specifies the activation mark of a bound service program. If the specified program activation does not exist then a *program activation not found* (hex 2C16) exception is signaled. If the *activation mark* does not correspond to a bound service program then an *invalid operation for program* (hex 2C15) exception is signaled.
- *ident type* (input) specifies how the export is identified.

0 =	reserved
1 =	by export ID. The item is identified by export identifier (or export ID.) The *number* operand specifies the export ID of the item. An export ID is the ordinal position, starting from 1, of the item in the bound service program's export list.
2 =	by name. The item is identified by name. The *name* operand provides the symbolic name of the item. The length of the name is specified by the *number* operand.
>2 =	reserved

  Use of a reserved value causes a *scalar value invalid* (hex 3203) exception to be signaled.

- *number* (input) specifies either the export ID or length of the character string which supplies the name.

- *name* (input) specifies the character string name. This operand is ignored if *ident type* is not 2, but must always be specified.
- *pointer* (*address* in, *value* out) specifies either a procedure pointer or space pointer to the exported item. If the requested export does not exist, 16 bytes of storage at the location indicated by this operand are set to binary 0.
- *export type* (*address* in, *value* out) identifies the type of export,

  0 = item not found

  1 = procedure export

  2 = data export

  3 = inaccessible data export

  If the item was not found the *pointer* is also set to binary 0.

The thread must have *execute* authority to the program specified by the *activation mark* in order to obtain the address of an exported item. In addition, a user state program may only materialize data exports from activations within the same activation group. An attempt to materialize a data export from another activation group will result in a value of 3 for *export type* and binary 0 being returned in the *pointer* operand. A system state program may materialize data exports from any activation.

Warning: Temporary Level 3 Header

Authorization Required

•

- Execute
  –
  – Program specified by *activation mark*.

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C15 Invalid Operation for Program

2C16 Program Activation Not Found

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Activation Group Attributes (MATAGPAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
MATAGPAT2 02C3	Receiver	Activation group mark	Attribute selection
MATAGPAT 02D3	Receiver	Activation group mark	Attribute selection

Operand 1: Space pointer.

Operand 2 for MATAGPAT2: Char(8) scalar.

Operand 2 for MATAGPAT: Unsigned binary(4) scalar.

Operand 3: Character(1) scalar.

Bound program access
Built-in number for MATAGPAT2 is 661. MATAGPAT2 (receiver : address activation_group_mark : address of unsigned binary(8) attribute_selection : address) OR Built-in number for MATAGPAT is 120. MATAGPAT (receiver : address activation_group_mark : address of unsigned binary(4) attribute_selection : address)

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note: The terms ″heap″ and ″heap space″ in this instruction refer to an ″activation group-based heap space″.

Note
It is recommended that you use the MATAGPAT2 instruction which supports 8-byte activation and activation group marks. The 8-byte *activation group mark* in the basic activation group attributes should also be used. 4-byte marks can wrap and produce unexpected results.

Description: This instruction will materialize the information selected by operand 3 for the activation group specified by operand 2 and return the information in the template supplied by operand 1. If the activation group mark specified by operand 2 is zero, then information about the activation group associated with the current invocation is returned. However, if the current invocation is associated with an activation which resides in a shared activation group owned by another process, or if no activation exists for the current invocation, then information about the default activation group with the same state as the invocation is returned.

In order to deal with the variable-length nature of some activation group attributes, the *attribute_selection* option is provided. All of the ″length-of-list″ type information can be gathered by selecting the first option described below.

Operand 3 can have the following values:

•
- Hex 00 — basic activation group attributes
- Hex 01 — activation group heap list option
- Hex 02 — program activation list option

  MATAGPAT2 returns a list of 8-byte activation marks. MATAGPAT returns a list of 4-byte activation marks.

Any value for operand 3 other than those listed will cause a *scalar value invalid* (hex 3203) exception.

The materialization template identified by operand 1 must be 16-byte aligned in the space. This materialization template has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Reserved (binary 0)	Char(8)	
16	10	Returned information	Char(*)	
*	*	—- End —-		

The **number of bytes provided** indicates the number of bytes provided for returned information by the user of the instruction. In all cases if the number of bytes provided is less than 8, then a *materialization length invalid* (hex 3803) exception will be signaled.

The **number of bytes available** is set by the instruction to indicate the actual number of bytes available to be returned. In no case does the instruction return more bytes of information than those available.

The format of **returned information** is described in the following paragraphs.

Basic Activation Group Attributes: The following information is returned when operand 3 is hex 00.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Root program	System pointer or Null	
32	20	Reserved (binary 0)	Char(16)	
48	30	Storage address recycling key	System pointer or Null	
64	40	Activation group name	Char(30)	
94	5E	Reserved (binary 0)	Char(2)	
96	60	Activation group mark	UBin(4)	
100	64	Reserved (binary 0)	Char(4)	
104	68	Heap space count	UBin(4)	
108	6C	Activation count	UBin(4)	
112	70	Static storage size	UBin(4)	
116	74	Reserved (binary 0)	UBin(4)	
120	78	Attributes	Char(1)	
120	78		Reserved	Bit 0
120	78		Activation group state	Bit 1
			0 = User	
			1 = System	
120	78		Is activation group named?	Bit 2
			0 = No	
			1 = Yes	
120	78		Destroy pending?	Bit 3
			0 = No	
			1 = Yes	
120	78		Shared activation group?	Bit 4
			0 = No	
			1 = Yes	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
120	78		Storage model	Bit 5
			0 = Single level storage	
			1 = Teraspace storage	
120	78		Reserved (binary 0)	Bits 6-7
121	79	Process access group (PAG) membership advisory attributes	Char(1)	
121	79		Reserved (binary 0)	Bit 0
121	79		Static storage	Bit 1
			0 = Do not create in PAG	
			1 = Permit creation in PAG	
121	79		Default heap storage	Bit 2
			0 = Do not create in PAG	
			1 = Permit creation in PAG	
121	79		Reserved (binary 0)	Bits 3-7
122	7A	Reserved (binary 0)	Char(6)	
128	80	Activation group mark	UBin(8)	
		For Non-Bound programs, the following datatype should be used:		
128	80		Activation group mark (Non-Bound program)	Char(8)
136	88	—- End —-		

Additional Description:

Root program
Those activation groups which are created by the machine (the default activation groups) do not have root programs, in which case this field is null.

Storage address recycling key
A system pointer is returned only if the *activation group state* is specified as *user*, otherwise the field is null.

Activation group name
For activation groups which do not have a symbolic name, this field contains all blanks.

Heap space count
This is the number of heap spaces currently associated with the activation group.

Activation count
This is the number of programs which are currently active within the activation group.

Static storage size
This is the maximum amount of static storage, in machine dependent units, which has been allocated to the activation group at any particular time. Note that this does not necessarily reflect the amount of storage currently in use.

Is activation group named?
Indicates whether the activation group is named or unnamed. The *activation group name* field contains blanks for unnamed activation groups. The default activation groups and those created with the "unnamed" attribute are unnamed.

Storage model
Indicates the storage model of the activation group. A *single level storage* activation group provides single level storage static storage to program activations while a *teraspace storage* activation group supplies teraspace storage.

Activation Group Heap List: When operand 3 is hex 01, the format of the *returned information* is an array of heap identifiers. This is a list of the heaps which are currently associated with the activation group. The *heap space count* (available in the basic template) indicates how many entries must be accommodated by the template. The format of the list is:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
16	10	Activation group heap list entry (repeated *heap space count* times)	[*] Bin(4)
*	*	—- End —-	

Information about a specific heap may be obtained from the Materialize Activation Group-Based Heap Space Attributes (MATHSAT) instruction.

Program Activation List: When operand 3 is hex 02, the format of the *returned information* is an array of *activation marks*. Each activation mark represents a program activation within the activation group. (The activation mark is a number which uniquely identifies the activation within a process.) The *activation count* (available in the basic template) indicates how many entries must be accommodated by the template. The format of the list is different for the MATAGPAT and MATAGPAT2 instructions.

Format of program activation list for MATAGPAT2:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
16	10	Program activation list entry (repeated *activation count* times) For Non-Bound programs, the following datatype should be used:	[*] UBin(8)	
16	10		Program activation list entry (Non-Bound program)	Char(8)
*	*	—- End —-		

Format of program activation list for MATAGPAT:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
16	10	Program activation list entry (repeated *activation count* times)	[*] UBin(4)
*	*	—- End —-	

Information about a specific activation may be obtained from the Materialize Activation Attributes (MATACTAT) instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1005 Authority Verification Terminated Due to Damaged Object
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded
- 1C04 Object Storage Limit Exceeded
- 1C09 Auxiliary Storage Pool Number Invalid

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid
- 2403 Pointer Addressing Invalid Object Type

2C Program Execution

- 2C13 Activation Group Not Found

32 Scalar Specification

- 3203 Scalar Value Invalid

38 Template Specification

- 3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Activation Group-Based Heap Space Attributes (MATHSAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
MATHSAT2 03E7	Materialize template	Heap identifier template	Attribute selection
MATHSAT 03B7	Materialize template	Heap identifier template	Attribute selection

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Character(1) scalar.

Bound program access

```
Built-in number for MATHSAT2 is 665.
MATHSAT2 (
         materialize_template       : address
         heap_identifier_template   : address
         attribute_selection        : address
)

OR
Built-in number for MATHSAT is 116.
MATHSAT (
         materialize_template       : address
         heap_identifier_template   : address
         attribute_selection        : address
)
```

Note: The term ″heap space″ in this instruction refers to an ″activation group-based heap space″.

Note

It is recommended that you use the MATHSAT2 instruction which supports 8-byte activation group marks. 4-byte activation group marks can wrap and produce unexpected results.

Description: This instruction will materialize the information selected by operand 3 for the heap space specified by operand 2 and return the selected information in the template indicated by operand 1.

Operand 3 can have three possible values:

•

- Hex 00 - Return heap space attributes
- Hex 01 - Return heap space attributes and mark information.
- Hex 02 - Return heap space attributes, mark information and allocation information.

Any value for operand 3 other than those listed will cause a *scalar value invalid* (hex 3203) exception.

The heap space attributes template identified by operand 1 must be 16-byte aligned in the space.

If operand 3 is equal to hex 00, then only the heap space attributes template information is returned. The format of the attributes template information is as follows (see the Create Activation Group-Based Heap Space (CRTHS) instruction for a description of these fields).

If operand 3 is equal to hex 02 AND the heap being used is a default heap, then not all the allocation information is available. Specifically, the following fields in the template will always be zero:

-
- *Current number of outstanding allocations*
- *Total number of frees*
- *Total number of allocations*
- *Maximum number of outstanding allocations*

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Maximum single allocation	UBin(4)	
12	C	Minimum boundary requirement	UBin(4)	
16	10	Creation size	UBin(4)	
20	14	Extension size	UBin(4)	
24	18	Domain	Bin(2)	
		Hex 0001 = The heap space domain is ″User″		
		Hex 8000 = The heap space domain is ″System″		
26	1A	Heap space creation options	Char(6)	
26	1A		Allocation strategy	Bit 0
			0 = Normal allocation strategy	
			1 = Force implicit space creation on each allocate	
26	1A		Heap space mark	Bit 1
			0 = Allow heap space mark	
			1 = Prevent heap space mark	
26	1A		Block transfer	Bit 2
			0 = Transfer the minimum storage transfer size for this object	
			1 = Transfer the machine default storage transfer size for this object	
26	1A		Process access group member	Bit 3
			0 = Do not create the heap space in the PAG	
			1 = Create the heap space in the PAG	
26	1A		Initialization allocations	Bit 4
			0 = Do not initialize allocations	
			1 = Initialize allocations	
26	1A		Overwrite freed allocations	Bit 5
			0 = Do not overwrite freed allocations	
			1 = Overwrite freed allocations	

Offset Dec	Hex	Field Name	Data Type and Length	
26	1A		Reserved (binary 0)	Bits 6-7
27	1B		Allocation value	Char(1)
28	1C		Freed value	Char(1)
29	1D		Reserved (binary 0)	Char(3)
32	20	Reserved (binary 0)	Char(64)	
96	60	Current number of outstanding allocations	UBin(4)	
100	64	Total number of reallocations	UBin(4)	
104	68	Total number of frees	UBin(4)	
108	6C	Total number of allocations	UBin(4)	
112	70	Maximum number of outstanding allocations	UBin(4)	
116	74	Size of the heap space in basic storage units	UBin(4)	
120	78	Number of outstanding marks	UBin(4)	
124	7C	Total number of extensions	UBin(4)	
128	80	—- End —-		

The first 4 bytes identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes a *materialization length invalid* (hex 3803) exception.

The second 4 bytes that are materialized identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the materialization length exception described previously) are signaled in the event that the receiver contains insufficient area for the materialization.

If operand 3 is equal to hex 01, then the *mark template information* is added to the *heap space attributes* template information. The *mark template information* is repeated for the number of outstanding marks. This information follows the *heap space attributes* template information. The format of the **mark template information** is as follows:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Mark template information	[*] Char(16)	
0	0		Mark identifier	Space pointer
*	*	—- End —-		

Given the list of mark identifiers with a mark identifier being entry N and an allocation belonging to mark identifier N, that allocation also belongs to mark identifier N-X, where X has values 1 to N-1 for all N>1.

If operand 3 is equal to hex 02, then the **allocation template** information is added to the *heap space attributes* and *mark template information*. The *allocation template* information is repeated for **current number of outstanding allocations**. This information follows the mark information template.

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Allocation template	[*] Char(48)	
0	0		Allocation address	Space pointer
16	10		Mark identifier	Space pointer

Offset Dec	Hex	Field Name	Data Type and Length	
32	20		Allocation size	UBin(4)
36	24		Reserved	Char(12)
*	*	—- End —-		

If **mark identifier** is null, this allocation is not associated with any mark. If it is not null it contains the most recent mark identifier to which the allocation belongs.

The **heap identifier template** identified by operand 2 must be 16-byte aligned in the space. The format of the template is different for the MATHSAT and MATHSAT2 instructions.

Format of heap identifier template for MATHSAT2 instruction:

Offset Dec	Hex	Field Name	Data Type and Length
0	0	Heap identifier template	Char(16)
0	0		Activation group mark identifier
			For Non-Bound programs, the following datatype should be used:
0	0		Activation group mark identifier (Non-Bound program)
8	8		Reserved (binary 0)
12	C		Heap identifier
16	10	—- End —-	

Format of heap identifier template for MATHSAT instruction:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Heap identifier template	Char(8)	
0	0		Activation group mark identifier	UBin(4)
4	4		Heap identifier	UBin(4)
8	8	—- End —-		

The **activation group mark identifier** may be zero, indicating the heap space specified by the **heap identifier** is in the current activation group. The value returned in the 4-byte activation group mark identifier may have wrapped.

MATHSAT will signal an *activation group access violation* (hex 2C12) exception if a program attempts to materialize heap space attributes of a heap space in an activation group to which the program does not have access.

Operands 1, 2 and 3 are not modified by the instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C09 Auxiliary Storage Pool Number Invalid

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2C Program Execution

2C12 Activation Group Access Violation

2C13 Activation Group Not Found

32 Scalar Specification

3203 Scalar Value Invalid

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

45 Heap Space

4501 Invalid Heap Identifier

4502 Invalid Request

4505 Heap Space Destroyed

4506 Invalid Heap Space Condition

Materialize Allocated Object Locks (MATAOL)

Op Code (Hex)	Operand 1	Operand 2
03FA	Receiver	Designated lockable item

Operand 1: Space pointer.

Operand 2: System pointer, object pointer or space pointer data object.

Bound program access

```
Built-in number for MATAOL is 48.
MATAOL (
        receiver                  : address
        designated_lockable_item  : address of system pointer OR
                                    address of object pointer OR
                                    address of space pointer(16)
)
```

Description: This instruction materializes the current allocated locks on a *designated lockable item*. If operand 2 is a system pointer, the current allocated locks on the object identified by the system pointer specified by operand 2 are materialized into the template specified by operand 1. If operand 2 is an object pointer, the current allocated locks on the specified object location are materialized into the template specified by operand 1. If operand 2 is a space pointer, the current allocated locks on the specified space location are materialized into the template specified by operand 1. The materialization template identified by operand 1 must be 16-byte aligned. The format of the materialization is as follows:

Offset				
Dec	Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Current cumulative lock status	Char(3)	
8	8		Lock states currently allocated (1 = yes)	Char(1)
8	8		LSRD	Bit 0

Offset Dec	Hex	Field Name	Data Type and Length	
8	8		LSRO	Bit 1
8	8		LSUP	Bit 2
8	8		LEAR	Bit 3
8	8		LENR	Bit 4
8	8		Locks implicitly set	Bit 5
8	8		Reserved (binary 0)	Bits 6-7
9	9		Reserved (binary 0)	Char(2)
11	B	Reserved (binary 0)	Char(1)	
12	C	Number of lock state descriptions	Bin(2)	
14	E	Reserved (binary 0)	Char(2)	
16	10	Lock state descriptions	[*] Char(32)	
		(repeated *number of lock state descriptions* times)		
16	10		Lock holder	System pointer
32	20		Lock state	Char(1)
			Hex 80 = LSRD lock request	
			Hex 40 = LSRO lock request	
			Hex 20 = LSUP lock request	
			Hex 10 = LEAR lock request	
			Hex 08 = LENR lock request	
			All other values are reserved	
33	21		Status of lock request	Char(1)
33	21		Lock scope object type	Bit 0
			0= Process control space	
			1= Transaction control structure	
33	21		Lock scope	Bit 1
			0= Lock is scoped to the *lock scope object type*	
			1= Lock is scoped to the thread	
33	21		Reserved (binary 0)	Bits 2-5
33	21		Implicit lock (machine applied)	Bit 6
33	21		Lock held by a process, thread or transaction control structure	Bit 7
34	22		Lock information	Char(1)
			A value of 1 in the corresponding bit indicates the condition is true:	
34	22		Reserved (binary 0)	Bits 0-5

Offset Dec	Offset Hex	Field Name	Data Type and Length	
34	22		Lock is held by some process, thread, or transaction control structure other than the current process or thread	Bit 6
34	22		Lock is held by the machine	Bit 7
35	23		Reserved (binary 0)	Char(1)
36	24		Unopened thread handle	UBin(4)
40	28		Thread ID	Char(8)
*	*	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than eight causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. Other than the *materialization length invalid* (hex 3803) exception, no exceptions are signaled should the receiver contain insufficient area for the materialization. As a result, the number of lock descriptions may be more than the number of lock descriptors that follow, since an insufficient area was provided to return the descriptors.

Locks may be implicitly applied by the machine (**implicit lock** is binary 1). If the implicit lock is held for a process or thread, a pointer to the associated process control space is returned in the **lock holder** field. If the implicit lock is held for a transaction control structure, a pointer to the associated transaction control structure is returned in the *lock holder* field. Locks held by the machine, but not related to a specific process, thread, or transaction control structure, cause the *lock holder*, **unopened thread handle**, and **thread ID** fields to each be assigned a value of binary 0.

When the lock is held by a process or a thread, the system security level is 40 or greater, and the invoker of this instruction is a user state program, then the *process control space* system pointer associated with the lock will be returned in the *lock holder* field if the lock is held by the current thread or its containing process. This field will be set to binary 0 if the lock is held by some other process or thread. When system security level 30 or less is in effect or when the invoking program is in system state, then the *lock holder* field will always be returned with the appropriate *process control space* system pointer value (which may be binary 0 if the machine holds the lock).

Locks may be held by a transaction control structure. If **lock scope object type** has a value of *transaction control structure*, then the *lock holder* field will contain a system pointer to the *transaction control structure* that holds the lock and the *unopened thread handle*, and *thread ID* fields will be assigned a value of binary 0.

The *lock information* will be set appropriately regardless of security level and program state.

Only a single lock state is returned for each *lock state description* entry.

A space pointer machine object cannot be specified for operand 2.

The **lock scope** field has no meaning if *lock held by a process, thread or transaction control structure* is binary 0.

A **lock state description** for a lock held by a process will have a value of binary 0 for the *unopened thread handle* and for the *thread ID*. A *lock state description* for a lock held by a thread will have a non-zero value for the *unopened thread handle* and for the *thread ID* to identify the specific thread within the process that is holding the lock.

When the invoker of this instruction is a user state program, then the *unopened thread handle* and *thread ID* fields will be returned if the lock is held by the current thread. These fields will be set to binary 0 if the lock is held by some other process or thread. When the invoking program is in system state, then the *unopened thread handle* and *thread ID* fields will always be returned with the appropriate values (which may be binary 0 if the machine holds the lock).

The maximum number of locks that can be materialized with this instruction is 32,767. No exception will be signaled if more than 32,767 exist and only the first 32,767 locks found will be materialized.

If operand 2 is a space pointer to a teraspace storage location, and the teraspace storage location is mapped, (from the current processes view), to a single level store location, then the locks on the single level store location will be materialized. If the teraspace storage location is not mapped, then locks on the teraspace storage location will be materialized.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Authority (MATAU)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0153	Receiver	System object	User profile or source template

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: System pointer or space pointer data object or null.

Bound program access

```
Built-in number for MATAU is 58.
MATAU (
       receiver                          : address
       system_object                     : address of system pointer
       user_profile_or_source_template   : address of system pointer OR
                                           address of space pointer(16) OR
                                           null operand
)
```

Description: This instruction materializes the specific types of authority for a *system object* available to the specified *user profile*. The private authorization that the *user profile* specified by operand 3 has to the permanent *system object* specified by operand 2, and the object's public authorization is materialized in operand 1. If operand 3 is null, then only the object's public authorization is materialized, and the *private authorization* field in the materialization is set to binary 0.

Except for certain special cases, the authority to be materialized is determined by first checking for direct authority to the object itself, then checking for indirect authority to the object through authority to an authority list containing the object. The first source of authority found is materialized and the source is indicated in the materialization.

The special case of the operand 3 *user profile* having *all object special authority* overrides any explicit private authorities that the *user profile* might hold to the object or its containing authority list and results in a materialization showing that the profile holds all private authorities directly to the object.

The special case of the operand 2 object being in an authority list which has the *override specific object authority* attribute in effect results in the authorization or lack of authorization held to the authority list completely overriding the explicit private authorities that the *user profile* might hold to the object. This case results in a materialization showing that the profile has just the private authorities it holds or doesn't hold to the authority list. That is, if the *user profile* has private authority to the object, but doesn't have private authority to the authority list, the materialization will show that the user does not have any private authority to the object. Similarly, if the *user profile* has both private authority to the object and to the authority list, the materialization will show that the user has only the private authority through the authority list. If operand 3 is null, then only the object's public authorization is materialized, and the private authorization field in the materialization is set to binary 0s.

Operand 3 may be specified as a system pointer which directly addresses the *user profile* to be checked as a source of authority or as a space pointer to a *source template* which identifies the source user profile. Specifying a template allows for additional controls over how the materialize operation is to be performed. The format of the *source template* is the following:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Source flags	Char(2)	
0	0		Ignore all object special authority	Bit 0

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = No	
			1 = Yes	
0	0		Reserved (binary 0)	Bits 1-15
2	2	Reserved (binary 0)	Char(14)	
16	10	User profile	System pointer	
32	20	—- End —-		

The **ignore all object special authority** source flag specifies whether special authority is to be ignored during the materialize operation. When *yes* is specified, just the explicitly held private authority that the specified user profile holds either directly to the object or indirectly to an authority list containing the object will be materialized. When *no* is specified, the authority provided by all object special authority, if held by the source user profile, is included and results in a materialization showing that the profile holds all private authorities directly to the object. *No* is the default for this flag value when the source template is not specified.

The **user profile** field specifies the user profile to be checked as a source of authority.

The format of the materialization (operand 1) is as follows:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization (contains a value of 16 for this instruction)	Bin(4)
8	8	Private authorization (1 = authorized)	Char(2)	
8	8		Object control	Bit 0
8	8		Object management	Bit 1
8	8		Authorized pointer	Bit 2
8	8		Space authority	Bit 3
8	8		Retrieve	Bit 4
8	8		Insert	Bit 5
8	8		Delete	Bit 6
8	8		Update	Bit 7
8	8		Ownership (1 = yes)	Bit 8
8	8		Excluded	Bit 9
8	8		Authority list management	Bit 10
8	8		Execute	Bit 11
8	8		Alter	Bit 12
8	8		Reference	Bit 13
8	8		Reserved (binary 0)	Bits 14-15
10	A	Public authorization (1 = authorized)	Char(2)	
10	A		Object control	Bit 0
10	A		Object management	Bit 1
10	A		Authorized pointer	Bit 2
10	A		Space authority	Bit 3
10	A		Retrieve	Bit 4
10	A		Insert	Bit 5

Offset Dec	Hex	Field Name	Data Type and Length	
10	A		Delete	Bit 6
10	A		Update	Bit 7
10	A		Reserved (binary 0)	Bit 8
10	A		Excluded	Bit 9
10	A		Authority list management	Bit 10
10	A		Execute	Bit 11
10	A		Alter	Bit 12
10	A		Reference	Bit 13
10	A		Reserved (binary 0)	Bits 14-15
12	C	Private authorization source	UBin(2)	
		0 = Authority to object		
		1 = Authority to authority list		
		2 = Authority to object via primary group		
		3 = Authority to authority list via primary group		
14	E	Public authorization source	UBin(2)	
		0 = Authority from object		
		1 = Authority from authority list		
16	10	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized (16 for this instruction). The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the materialization length exception) are signaled in the event that the receiver contains insufficient area for the materialization.

Any of the four authorizations- **retrieve**, **insert**, **delete**, or **update**-constitute operational authority.

If this instruction references a temporary object, all public authority states are materialized. Private authority states are not materialized.

Warning: Temporary Level 3 Header

Authorization Required

•

- Operational
  –
  – Operand 3
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize
  –
  – Operand 2
  – Operand 3
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Authority List (MATAL)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
01B3	Receiver	Authority list or Authority list extension	Materialization options

Operand 1: Space pointer

Operand 2: System pointer or open pointer

Operand 3: Space pointer

Bound program access

```
Built-in number for MATAL is 59.
MATAL (
       receiver                 : address
       authority_list           : address of system pointer OR
                                  address of open pointer
       materialization_options  : address
)
```

Description: Based on the contents of the *materialization options* specified by operand 3, the symbolic identification and/or system pointers to all, or a selected set, of the objects contained in the authority list or authority list extension specified by operand 2 are materialized into the *receiver* specified by operand 1.

A space pointer machine object may not be specified for operand 2.

The *materialization options* operand has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization control	Char(2)	
0	0		Information requirements	Char(1)
			Hex 12 = Materialize count of entries matching the criteria	
			Hex 22 = Materialize identification of entries matching the criteria and return information using short description format	
			Hex 32 = Materialize identification of entries matching the criteria and return information using long description format	
1	1		Selection criteria	Char(1)
			Hex 00 = All authority list or authority list extension entries	
			Hex 01 = Type code selection	
			Hex 02 = Type code/subtype code selection	
2	2	Reserved (binary 0)	Bin(2)	
4	4	Type code	Char(1)	
5	5	Subtype code	Char(1)	
6	6	Reserved (binary 0)	Char(30)	
36	24	—- End —-		

The **information requirements** field specifies the type of materialization, just a *count of entries*, *short descriptions*, or *long descriptions*, which is being requested.

The **selection criteria** field specifies the criteria to be used in selecting the authority list or authority list extension entries for which information is to be presented. The **type code** and **subtype code** fields contain the selection criteria when a selective materialization is specified.

When *type code* or *type/subtype codes* are part of the selection criteria, only entries that have the specified codes are considered.

The format of the materialization (operand 1) is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Materialization size specification	Char(8)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Authority list identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Authority list creation options	Char(4)	
40	28		Existence attributes	Bit 0
			1 = Permanent (always permanent)	
40	28		Space attribute	Bit 1
			0 = Fixed length	
			1 = Variable length	
40	28		Reserved (binary 0)	Bits 2-31
44	2C	Reserved (binary 0)	Char(4)	
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
57	39	Reserved	Char(7)	
64	40	Context	System pointer	
80	50	Reserved	Char(16)	
96	60	Authority list attributes	Char(4)	
96	60		Override specific object authority	Bit 0
			0 = No	
			1 = Yes	
96	60		Reserved (binary 0)	Bits 1-31
100	64	Reserved (binary 0)	Char(28)	
128	80	Entries header	Char(16)	
128	80		Number of entries available	UBin(4)
132	84		Reserved	Char(12)
144	90	—- End —-		

If no description (*information requirements* = hex 12) is requested in the *materialization options* operand, the above constitutes the information available for materialization. If a description (short or long) is requested by the *materialization options* operand, a description entry is present (assuming a sufficient size *receiver*) for each object materialized into the *receiver*. Either of the following entry formats may be selected.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Short description entry	Char(32)	
0	0		Type code	Char(1)
1	1		Subtype code	Char(1)
2	2		Reserved	Char(14)
16	10		System object	System pointer
32	20	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Long description entry	Char(128)	
0	0		Type code	Char(1)
1	1		Subtype code	Char(1)
2	2		Object name	Char(30)
32	20		Reserved	Char(16)
48	30		System object	System pointer
64	40		Object owning user profile	System pointer
80	50		Context	Char(48)
80	50		Type code	C
81	51		Subtype code	C
82	52		Context name	C
112	70		Context pointer	S
128	80	—- End —-		

The first four bytes of the materialization output identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes and pointers as can be contained in the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested for materialization, the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception signaled above.

The creation attributes (bytes 40 through 127) are not returned when materializing an authority list extension. These fields of the template will be set to hex zero. The fields are:

- *Authority list creation options*
- *Size of space*
- *Initial value of space*
- *Performance class*
- *Context*
- *Authority list attributes*

When an authority list extension is materialized, a null pointer value will be returned for *object owning user profile*.

The **number of entries available** field specifies the number of authority list entries which satisfied the selection criteria and were therefore materialized. A value of zero indicates no entries were available.

The object identification information (in the short and long description entries), if requested by the *materialization options* operand, is present for each entry in the authority list or authority list extension that satisfies the search criteria.

The object pointer information (in the long description entry only), if requested by the *materialization options* operand, is present for each entry in the authority list or authority list extension that satisfies the search criteria.

If the object addressed by the system pointer is not addressed by a context, the *context type* field is set to hex 00 or if the object is addressed by a machine context, the *context type* field is set to hex 81. Additionally, in either of these cases, the *context pointer* is set to a null pointer value.

Warning: Temporary Level 3 Header

Authorization Required

•

• Retrieve

Operand 2

• All object special authority

  –

  – Operand 2 if authority list extension

Lock Enforcement

•

• Materialization

Operand 2

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

0A04 Special Authorization Required

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Materialize Authorized Objects (MATAUOBJ)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
013B	Receiver	User profile or User profile extension	Materialization options/template

Operand 1: Space pointer

Operand 2: System pointer or open pointer

Operand 3: Character scalar

Bound program access

```
Built-in number for MATAUOBJ is 60.
MATAUOBJ (
          receiver                 : address
          user_profile             : address of system pointer OR
                                     address of open pointer
          materialization_options  : address
)
```

Description: This instruction materializes the identification and the system pointers to all or selected system objects that are privately owned and/or authorized by a specified *user profile* or *user profile extension*, and/or for which the profile is the primary group. For the *user profile* or *user profile extension* (operand 2), the *materialization options* (operand 3) specify object selection criteria and the format and location of the object materialization data. The *receiver* space (operand 1) always indicates the number of objects materialized, and contains the object materialization data unless the *materialization options* specify an independent index to contain the data.

A space pointer machine object may not be specified for operand 2.

When *format of operand 3* is *operand 3 is a Char(1) scalar*, operand 3 is viewed as a Char(1) scalar. This option does not permit object selection by type and subtype, does not allow a continuation point to be specified, and returns all object materialization data in the *receiver* (operand 1). The short template header format is used. Following are the valid operand 3 values which may be used with the short template header format (operand 1):

Value (hex)	Meaning
07	Verify user profile integrity for all authorized, owned objects, and objects for which profile is the primary group.
11	Materialize count of owned objects.
12	Materialize count of authorized objects.
13	Materialize count of all authorized and owned objects.
14	Materialize count of objects for which profile is the primary group.
15	Materialize count of owned objects and objects for which profile is the primary group.
16	Materialize count of authorized objects and objects for which profile is the primary group.
17	Materialize count of all authorized, owned objects, and objects for which profile is the primary group.
21	Materialize identification of owned objects using short description entry format.
22	Materialize identification of authorized objects using short description entry format.
23	Materialize identification of all authorized and owned objects using short description entry format.
24	Materialize identification of objects for which profile is the primary group using short description entry format.
25	Materialize identification of owned objects and objects for which profile is the primary group using short description entry format.

Value (hex)	Meaning
26	Materialize identification of authorized objects and objects for which profile is the primary group using short description entry format.
27	Materialize identification of all authorized, owned objects, and objects for which profile is the primary group using short description entry format.
31	Materialize identification of owned objects using long description entry format.
32	Materialize identification of authorized objects using long description entry format.
33	Materialize identification of all authorized and owned objects using long description entry format.
34	Materialize identification of objects for which profile is the primary group using long description entry format.
35	Materialize identification of owned objects and objects for which profile is the primary group using long description entry format.
36	Materialize identification of authorized objects and objects for which profile is the primary group using long description entry format.
37	Materialize identification of all authorized, owned objects, and objects for which profile is the primary group using long description entry format.

Usage note:

Although damage in a user profile is extremely rare, option hex 07 can be used to verify the integrity of the user profile without referencing any owned or authorized objects. This allows the caller to quickly verify a user profile before a lengthy procedure, such as a product install. If any integrity problems are found, the user profile will be damaged and a *system object damage state* (hex 1004) exception will be signaled.

Following are the valid operand 3 values which may be used with the long template header format (operand 1):

Value (hex)	Meaning
51-57	These long template header materialization options are the same as the short template header materialization options 11-17 (hex).
61-67	These long template header materialization options are the same as the short template header materialization options 21-27 (hex).
71-77	These long template header materialization options are the same as the short template header materialization options 31-37 (hex) except that the context extension is materialized for each object as well.

When *format of operand 3* is set to *operand 3 is variable length*, operand 3 is viewed as variable-length, must be 16-byte aligned in the space, and has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Materialization options Valid values are hex 91-97, A1-A7, B1-B7, D1-D7, E1-E7, and F1-F7. They have the same meanings as the corresponding values with *format of operand 3* set to binary 0 (i.e., hex 91 has the same meaning as hex 11).	Char(1)
0	0	Format of operand 3 **0 =** Operand 3 is a Char(1) scalar **1 =** Operand 3 is variable length	Bit 0

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0		Materialization identifier	Bits 1-7
1	1	Materialization flags	Char(1)	
1	1		Restrict information scope This is an input bit which only has meaning when materialization data is being returned in the operand 1 *receiver* template. When there is more data to be materialized than can be contained in the template, then when this bit is set to binary 1, the *number of bytes available for materialization*, the *number of objects owned by user profile*, the *number of objects for which the profile is the primary group*, and the *number of objects privately authorized to user profile* output fields are restricted to reflect only the information returned in the template; when set to binary 0, the output fields reflect the total amount of materialization data available, even though the template may not be large enough to contain it all.	Bit 0
1	1		More materialization data available This output bit has meaning only when materialization data is being returned in the operand 1 receiver template. When set to binary 1, it indicates that objects exist beyond those for which materialization data was returned in the template; when set to binary 0 it indicates the end of the objects was reached.	Bit 1
1	1		Continuation point specified This is an input bit. When set to binary 1, it indicates that a continuation point is specified in the *continuation point* field; when set to binary 0, continuation processing is ignored.	Bit 2
1	1		Avoid storage correction This is an input bit. When set to binary 1, it indicates that storage correction is avoided on owned objects. When set to binary 0, storage correction is performed as required.	Bit 3
1	1		Reserved (binary 0)	Bits 4-7
2	2	Reserved (binary 0)	Char(30)	
32	20	Independent Index pointer If the pointer does not exist, the instruction returns all object materialization data in the receiver (operand 1). Otherwise it returns only the template header in the receiver and returns the object materialization data in the independent index.	System pointer	
48	30	Continuation point If the *continuation point specified* bit is *on*, when the instruction begins, if this field contains a system pointer or the storage form of a system pointer, then materialization data is returned for objects found in the profile following the object identified by the continuation point; otherwise, materialization data is returned beginning with the object which is logically first.	Char(16) or System pointer	
64	40	Object type/subtype range array	Bin(2)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
		Indicates the number of *object type/subtype range*s specified in the array immediately following. If zero, objects of all types and subtypes are materialized. If larger than zero, only objects included in one or more of the *type/subtype ranges* specified in the array are materialized.		
66	42	Object type/subtype array	[*] Char(4)	
		An array of object type/subtype ranges qualifying the objects materialized. Each array element represents a range of *object type/subtypes* and has the following format:		
66	42		Start of range	Char(2)
66	42		Object type code	Char(1)
67	43		Object subtype code	Char(1)
68	44		End of range	Char(2)
68	44		Object type code	Char(1)
69	45		Object subtype code	Char(1)
*	*	—- End —-		

All *materialization options* with the low order bit on (except for option hex 07) also verify and correct the user profile's system storage utilization (storage used on the system ASP and basic ASPs) and varied on independent ASP storage utilization. The corrected storage utilization is not returned by MATAUOBJ and the MATUP instruction must be used to obtain the storage utilization values. The storage utilization is *not* corrected if either of the following are true:

•
- The extended form of operand 3 is used and a valid continuation point is specified, or
- The *avoid storage correction* field is set to binary 1.

The **avoid storage correction** field indicates whether or not system storage utilization for the user profile should be verified or corrected at this time. This may improve performance significantly when *avoid storage correction* is set to binary 1.

The order of materialization is owned objects, objects privately authorized to the *user profile*, and objects for which the profile is the primary group (as specified in the *materialization options* operand). No authorizations are stored in the system pointers that are returned.

The template identified by operand 1 must be 16-byte aligned in the space. For options hex 07 through hex 37 and hex 91 through hex B7, the **short template header** is materialized. It has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Number of objects owned by user profile	Bin(2)	
10	A	Number of objects privately authorized to user profile	Bin(2)	
12	C	Number of objects for which the user profile is the primary group	Bin(2)	
14	E	Reserved (binary 0)	Char(2)	
16	10	—- End —-		

For options hex 51 through 77 and hex D1 through hex F7, the **long template header** is materialized. It has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Number of objects owned by user profile	Bin(4)	
12	C	Number of objects privately authorized to user profile	Bin(4)	
		The following header information is only provided when an option requesting primary group is selected (for example hex 54 - 57, 64 - 67 and 74 - 77 etc).		
16	10	Number of objects for which the profile is the primary group	Bin(4)	
20	14	Reserved (binary 0)	Char(12)	
32	20	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the *receiver* contains insufficient area for the materialization. If the *restrict information scope* flag is binary 1, then the field contains the number of bytes materialized, rather than the number of bytes available to be materialized. The **number of objects owned by user profile**, the **number of objects for which the profile is the primary group**, and the **number of objects privately authorized to user profile** will contain the appropriate counts for each type of authorized/owned object. For options greater than hex 07, the authorized/owned objects are verified and these counts are corrected. For option hex 07, the counts are not verified.

If the **restrict information scope** flag is binary 1, then the *number of objects owned by user profile* and the *number of objects privately authorized by user profile* fields reflect the number of objects for which complete materialization data is returned, rather than the total number of such objects.

If no description is requested in the *materialization options* field, the above constitutes the information available for materialization. If a description (short, long, or long with context extension) is requested by the *materialization options* field, a description entry is present for each object materialized into the *receiver* (assuming it is of sufficient size) or into the independent index. Object materialization data is in one of the following formats depending on the *materialization options* and the object into which it is materialized:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Short description entry materialized into receiver	Char(32)	
0	0		Object type code	Char(1)
1	1		Object subtype code	Char(1)
2	2		Private authorization	Char(2)
4	4		Reserved (binary 0)	Char(10)
14	E		Independent ASP number	Char(2)
16	10		Object pointer	System pointer
32	20	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Long description entry materialized into receiver	Char(64)	
0	0		Object type code	Char(1)
1	1		Object subtype code	Char(1)
2	2		Object name	Char(30)
32	20		Private authorization	Char(2)
34	22		Public authorization	Char(2)
36	24		Reserved (binary 0)	Char(10)
46	2E		Independent ASP number	Char(2)
48	30		Object pointer	System pointer
64	40	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Long description entry with context extension materialized into receiver	Char(112)	
0	0		Object type code	Char(1)
1	1		Object subtype code	Char(1)
2	2		Object name	Char(30)
32	20		Private authorization	Char(2)
34	22		Public authorization	Char(2)
36	24		Reserved (binary 0)	Char(10)
46	2E		Independent ASP number	Char(2)
48	30		Object pointer	System pointer
64	40		Context type code	Char(1)
65	41		Context subtype code	Char(1)
66	42		Context name	Char(30)
96	60		Context pointer	System pointer
112	70	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Short description entry materialized into independent index	Char(32)	
0	0		Entry type code Hex 40 = Owned object Hex 80 = Authorized object Hex A0 = Profile is primary group of object	Char(1)
1	1		Object type code	Char(1)
2	2		Object subtype code	Char(1)
3	3		Private authorization	Char(2)
5	5		Reserved (binary 0)	Char(9)
14	E		Independent ASP number	Char(2)
16	10		Object pointer	System pointer
32	20	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Long description entry materialized into independent index	Char(64)	
0	0		Entry type code	Char(1)
			Hex 40 = Owned object	
			Hex 80 = Authorized object	
			Hex A0 = Profile is primary group of object	
1	1		Object type code	Char(1)
2	2		Object subtype code	Char(1)
3	3		Object name	Char(30)
33	21		Private authorization	Char(2)
35	23		Reserved (binary 0)	Char(2)
37	25		Public authorization	Char(2)
39	27		Reserved (binary 0)	Char(7)
46	2E		Independent ASP number	Char(2)
48	30		Object pointer	System pointer
64	40	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Long description entry with context extension materialized into independent index	Char(112)	
0	0		Entry type code	Char(1)
			Hex 40 = Owned object	
			Hex 80 = Authorized object	
			Hex A0 = Profile is primary group of object	
1	1		Context type code	Char(1)
2	2		Context subtype code	Char(1)
3	3		Context name	Char(30)
33	21		Object type code	Char(1)
34	22		Object subtype code	Char(1)
35	23		Object name	Char(30)
65	41		Private authorization	Char(2)
67	43		Reserved (binary 0)	Char(2)
69	45		Public authorization	Char(2)
71	47		Reserved (binary 0)	Char(7)
78	4E		Independent ASP number	Char(2)
80	50		Object pointer	System pointer
96	60		Context pointer	System pointer
112	70	—- End —-		

Following is the format of the authorization information:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Private authorization	Char(2)

Offset Dec	Hex	Field Name	Data Type and Length	
		(1 = authorized)		
0	0		Object control	Bit 0
0	0		Object management	Bit 1
0	0		Authorized pointer	Bit 2
0	0		Space authority	Bit 3
0	0		Retrieve	Bit 4
0	0		Insert	Bit 5
0	0		Delete	Bit 6
0	0		Update	Bit 7
0	0		Ownership (1 = yes)	Bit 8
0	0		Excluded	Bit 9
0	0		Authority list management	Bit 10
0	0		Execute	Bit 11
0	0		Alter	Bit 12
0	0		Reference	Bit 13
0	0		Reserved (binary 0)	Bits 14-15
2	2	Public authorization	Char(2)	
		(1 = authorized)		
2	2		Object control	Bit 0
2	2		Object management	Bit 1
2	2		Authorized pointer	Bit 2
2	2		Space authority	Bit 3
2	2		Retrieve	Bit 4
2	2		Insert	Bit 5
2	2		Delete	Bit 6
2	2		Update	Bit 7
2	2		Reserved (binary 0)	Bit 8
2	2		Excluded	Bit 9
2	2		Authority list management	Bit 10
2	2		Execute	Bit 11
2	2		Alter	Bit 12
2	2		Reference	Bit 13
2	2		Reserved (binary 0)	Bits 14-15
4	4	—- End —-		

When context information is materialized, if the object addressed by the system pointer is not addressed by a context, the *context type* field is set to hex 00 or if the object is addressed by the machine context, the *context type* field is set to hex 81. Additionally, in either of these cases, the *context pointer* is set to the system default pointer does not exist value.

When the **more materialization data available** flag is binary 1, the pointer to the object within the last entry in the operand 1 *receiver* template may be specified as the continuation point on a subsequent invocation of this instruction, to cause materialization to continue, starting with the "logically next" object. To determine whether the continuation point is within the owned or authorized objects, the *ownership* bit in the private authorizations of the last materialized object may be tested. This instruction does not guarantee an atomic snapshot of the *user profile* or *user profile extension* across a continuation request.

The following considerations apply when object materialization data is returned in an independent index:

•

- System pointers returned in index entries are not set unless the index is created to contain both pointer and scalar data.
- Entry data may be truncated or padded on the right with hex zeroes to conform to the index's key and/or fixed entry lengths.
- An entry is added to the index for each qualifying object. Previously existing entries which are thereby duplicated are replaced.
- In order to ensure that index entries inserted within the same execution of this instruction are not duplicates of each other, the *index entry length* (if fixed) and *key length* (if keyed) must be sufficiently large to include the object pointer within the entry data.

Authorization Required

•
- Operational
  –
  – Operand 2
- All object special authority
  –
  – Operand 2 if user profile extension
- Execute
  –
  – Contexts referenced for address resolution
- Retrieve
  –
  – Operand 2 if materializing owned objects
- Insert
  –
  – Independent index if identified by operand 3

Lock Enforcement

•
- Materialize
  –
  – Contexts referenced for address resolution
  – Operand 2 if materializing owned objects
- Modify
  –
  – Independent index if identified by operand 3

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

0A04 Special Authorization Required

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2204 Object Not Eligible for Operation

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Authorized Users (MATAUU)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0143	Receiver	System object	Materialization options

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: Character(1) scalar.

Bound program access

```
Built-in number for MATAUU is 61.
MATAUU (
        receiver                : address
        system_object           : address of system pointer
        materialization_options   : address
)
```

Description: The instruction materializes the authorization states and the identification of the user profile(s). The *materialization options* (operand 3) for the *system object* (operand 2) are returned in the *receiver* (operand 1). The *materialization options* for operand 3 have the following format:

Value (Hex)	Meaning
11	Materialize public authority.
12	Materialize public authority and number of privately authorized profiles.
21	Materialize identification of owning profile using short description entry format.
22	Materialize identification of privately authorized profiles using short description entry format.
23	Materialize identification of owning and privately authorized profiles using short description entry format.

Value (Hex)	Meaning
24	Materialize identification of primary group profile using short description entry format.
25	Materialize identification of owning profile and primary group profile using short description entry format.
26	Materialize identification of privately authorized profiles and primary group profile using short description entry format.
27	Materialize identification of owning profile, primary group profile, and privately authorized profiles using short description entry format.
31	Materialize identification of owning profile using long description entry format.
32	Materialize identification of privately authorized profiles using long description entry format.
33	Materialize identification of owning and privately authorized profiles using long description entry format.
34	Materialize identification of primary group profile using long description entry format.
35	Materialize identification of owning profile and primary group profile using long description entry format.
36	Materialize identification of privately authorized profiles and primary group profile using long description entry format.
37	Materialize identification of owning profile, primary group profile, and privately authorized profiles using long description entry format.

The order of materialization is an entry for the owning user profile, an entry for the primary group profile, followed by a list (0 to n entries) of entries for user profiles having private authorization to the object (as specified in the *materialization options* operand). The authorization field within the system pointers will not be set.

If the primary group profile for the object is not set (there is no primary group for the object), and the primary group profile is requested by the *materialization options* operand, the entry for the primary group profile will be set to binary 0.

The template identified by operand 1 must be 16-byte aligned in the space and has the following format:

Offset				
Dec	Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Public authorization (1 = authorized)	Char(2)	
8	8		Object control	Bit 0
8	8		Object management	Bit 1
8	8		Authorized pointer	Bit 2
8	8		Space authority	Bit 3
8	8		Retrieve	Bit 4
8	8		Insert	Bit 5
8	8		Delete	Bit 6
8	8		Update	Bit 7
8	8		Reserved (binary 0)	Bit 8
8	8		Excluded	Bit 9
8	8		Authority list management	Bit 10
8	8		Execute	Bit 11
8	8		Alter	Bit 12
8	8		Reference	Bit 13
8	8		Reserved (binary 0)	Bits 14-15
10	A	Number of privately authorized user profiles	Bin(2)	
12	C	Reserved (binary 0)	Char(4)	

Offset Dec	Hex	Field Name	Data Type and Length
16	10	—- End —-	

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the *receiver* contains insufficient area for the materialization.

If no description is requested by the *materialization options* field, the template identified by operand 1 constitutes the information available for materialization. If a description (short or long) is requested by the *materialization options* field, a description entry is present (assuming there is a sufficient sized *receiver*) for each user profile materialized or available to be materialized into the *receiver*. Either of the following entry types may be selected.

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Short description entry	Char(32)	
0	0		User profile type code	Char(1)
1	1		User profile subtype code	Char(1)
2	2		Private authorization (1 = authorized)	Char(2)
2	2		Object control	Bit 0
2	2		Object management	Bit 1
2	2		Authorized pointer	Bit 2
2	2		Space authority	Bit 3
2	2		Retrieve	Bit 4
2	2		Insert	Bit 5
2	2		Delete	Bit 6
2	2		Update	Bit 7
2	2		Ownership (1 = yes)	Bit 8
2	2		Excluded	Bit 9
2	2		Authority list management	Bit 10
2	2		Execute	Bit 11
2	2		Alter	Bit 12
2	2		Reference	Bit 13
2	2		Reserved (binary 0)	Bits 14-15
4	4		Reserved (binary 0)	Char(12)
16	10		User profile	System pointer
32	20	—- End —-		

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Long description entry	Char(64)	
0	0		User profile type code	Char(1)
1	1		User profile subtype code	Char(1)
2	2		User profile name	Char(30)
32	20		Private authorization	Char(2)

Offset Dec	Hex	Field Name	Data Type and Length	
			(1 = authorized)	
32	20		Object control	Bit 0
32	20		Object management	Bit 1
32	20		Authorized pointer	Bit 2
32	20		Space authority	Bit 3
32	20		Retrieve	Bit 4
32	20		Insert	Bit 5
32	20		Delete	Bit 6
32	20		Update	Bit 7
32	20		Ownership (1 = yes)	Bit 8
32	20		Excluded	Bit 9
32	20		Authority list management	Bit 10
32	20		Execute	Bit 11
32	20		Alter	Bit 12
32	20		Reference	Bit 13
32	20		Reserved (binary 0)	Bits 14
34	22		Reserved (binary 0)	Char(14)
48	30		User profile	System pointer
64	40	—- End —-		

If this instruction references a temporary object, all public authority states are materialized. The privately authorized user, primary group, and owner profile(s) descriptions are not materialized (binary 0).

Warning: Temporary Level 3 Header

Authorization Required

•

- Execute
  –
  – Contexts referenced for address resolution
- Object management or ownership
  –
  – Operand 2 object (when object is not an authority list)
- Authority list management or ownership
  –
  – Operand 2 object (when object is an authority list)

Lock Enforcement

•

- Materialize
  –
  – Operand 2
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Bound Program (MATBPGM)

Op Code (Hex)	Operand 1	Operand 2
02C6	Materialization request template	Bound program or bound service program

Operand 1: Space pointer.

Operand 2: System pointer.

Bound program access

```
Built-in number for MATBPGM is 109.
MATBPGM (
         materialization_request_template         : address
         bound_program_or_bound_service_program   : address of system pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note: The terms ″heap″ and ″heap space″ in this instruction refer to an ″activation group-based heap space″.

Description: The *bound program or bound service program* identified by operand 2 is materialized according to the specifications provided by operand 1.

Operand 2 is a system pointer that identifies the *bound program or bound service program* to be materialized. If operand 2 does not refer to a program object, a *pointer addressing invalid object type* (hex 2403) exception will be signaled. If operand 2 refers to a program, but not to a bound program or bound service program, then a *program not eligible for operation* (hex 220A) exception will be signaled.

The values in the materialization relate to the current attributes of the materialized bound program. Components are the materializable parts of a bound program or bound service program. Components may not be available for materialization because they were not encapsulated during bound program creation. Other components may not be available for materialization because they contain no data.

This instruction does not process teraspace addresses used for its operands, nor used in any space pointer contained in a template. Any teraspace address use will cause an *unsupported space use* (hex 0607) exception to be signaled, whether the issuing program is teraspace capable or not.

Note: The bound program *materialization request template* takes the form of a variable length array of materialization requests.

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Template size specification	Char(8)
0	0		Number of bytes provided
4	4		Reserved (binary 0)
8	8	Number of materialization requests	UBin(4)
12	C	Reserved (binary 0)	Char(4)
16	10	Array of materialization requests	[*] Char(32)
16	10		Receiver
32	20		Bound program materialization options
32	20		General bound program materialization options
32	20		General bound pro
			0 = Do not n
			1 = Materiali
32	20		Reserved
			Note: Reserved
32	20		Program copyright
			0 = Do not n
			1 = Materiali
32	20		Bound service prog
			0 = Do not n
			1 = Materiali
32	20		Bound modules in
			0 = Do not n
			1 = Materiali
32	20		Bound program str
			0 = Do not n
			1 = Materiali
32	20		Bound program lin
			0 = Do not n
			1 = Materiali
32	20		Reserved
			Note: Reserved
32	20		Activation group d
			0 = Do not n
			1 = Materiali
32	20		Activation group d

Offset Dec	Offset Hex	Field Name	Data Type and Length
			0 = Do no
			1 = Mater
32	20		Reserved (binar
34	22		Specific bound program materialization options
34	22		Specific bound
			0 = Do no
			1 = Mater
34	22		Reserved (binar
35	23		Specific bound service program materialization options
35	23		Reserved (binar
35	23		Signatures infor
			0 = Do no
			1 = Mater
35	23		Exported progr
			0 = Do no
			1 = Mater
35	23		Exported progr
			0 = Do no
			1 = Mater
35	23		Reserved (binar
36	24		Bound module materialization options
36	24		General module information
			0 = Do not materialize
			1 = Materialize
36	24		Reserved
			Note: Reserved for IBM Internal Use Only. If used, u
36	24		Module string directory component
			0 = Do not materialize
			1 = Materialize
36	24		Reserved
			Note: Reserved for IBM Internal Use Only. If used, u
36	24		Reserved (binary 0)
36	24		Module copyright strings
			0 = Do not materialize
			1 = Materialize
36	24		Reserved
			Note: Reserved for IBM Internal Use Only. If used, u
36	24		Reserved (binary 0)
40	28		Bound module materialization number
44	2C		Reserved (binary 0)
*	*	—- End —-	

Description of bound program materialization request template fields: Each of the reserved fields must be set to binary 0s, or a *template value invalid* (hex 3801) exception will be signaled.

Number of bytes provided
This is the size in bytes of the materialization request template. If this size does not correspond to the actual number of bytes in the materialization request template, then a *template value invalid* (hex 3801) exception will be signaled. This does not include any storage for returned data. That storage is pointed to by the *receiver* values.

Number of materialization requests
The number of requests in the array of materialization requests is specified by this value. If this number is greater than the actual number of materialization requests provided, then a *template value invalid* (hex 3801) exception will be signaled.

Materialization requests

This is an array of materialization requests. A materialization request consists of one or more bits, and an optional module number specified to be materialized into the corresponding *receiver*. Each materialization request consists of the following fields.

Receiver

This is a pointer to a space which will hold the materialized data. The space pointed to must be aligned on a 16-byte boundary, and must be at least 8 bytes long. This is so that it can hold the *bytes provided* and *bytes available* field of the *receiver*. If the space is not at least 8 bytes long a *template value invalid* (hex 3801) exception will be signaled.

Bound program materialization options

This bit mapped field specifies the parts of the bound program object to be materialized. A materialization request need not specify any program materialization options. If no bits are set, a bit must be set in the *bound module materialization options* field, or a *template value invalid* (hex 3801) exception will be signaled. Multiple options may be specified. When multiple options are specified, all of the requested data will be materialized into one receiver. The pieces requested on the materialization will be placed in the receiver in the order that the option bits are defined. If options are also specified on the *bound module materialization options* field, the materialized data for those options will follow that data materialized for the *bound program materialization options*.

The *bound program materialization options* are split into three distinct materialization bit sets.

1. The **general bound program materialization options** contains bits that represent data that can be materialized for either bound programs or bound service programs.
2. The **specific bound program materialization options** contains bits that represent data that can be materialized only for bound programs, and not for bound service programs.
3. The **specific bound service program materialization options** contains bits that represent data that can be materialized only for bound service programs.

If a bit is on to materialize information that is not contained in the type of bound program being materialized, then an indication that the information is not materializable will be provided in the receiver header. No exception, in this case, will be signaled.

Each of the requested pieces will be placed on a 16-byte boundary within the *receiver*.

The **general bound program information** field specifies that general information about the bound program object should be

Format of materialized data:

Format of Receiver:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Number of bytes provided for materialization	UBin(4)
4	4	Number of bytes available for materialization	UBin(4)
8	8	Reserved (binary 0)	Char(8)
16	10	Materialized data	Char(*)
*	*	—- End —-	

Bytes provided

This is the number of bytes the user is providing to hold the materialized data. It must be greater than or equal to eight. If it is equal to eight, then no data will actually be materialized, and the number of bytes required to materialize the requested data will be placed in *bytes available*. If the value provided is greater than eight, but less than the number of bytes required to hold the requested data, the data will be truncated and no exception will be signaled. Note that a value greater than eight, but less than 16 will result in no data being materialized, since bytes 9-16 are reserved.

If the receiver is smaller than the size indicated by *bytes provided*, and the materialized data is larger than the space provided in *receiver*, the *space addressing violation* (hex 0601) exception will be signaled unless *receiver* is an automatically extendable space object. If *receiver* is an automatically extendable space object, the space will be extended, up to its maximum size.

Bytes available

If *bytes provided* is greater than eight, this contains the number of bytes that have been used for the materialization, including any reserved bytes or bytes used for padding. If *bytes provided* is equal to eight, this contains the total size of the *receiver* needed to hold the requested materialization. A value of zero is returned if there is no data to materialize.

Materialized data

For each bit on in the *bound program materialization options* and *bound module materialization options*, this will contain the associated data. Each entry will be preceded by a common header which identifies the type of data and the offset to the next entry. When multiple bits are on in the same request, the data is returned in the order defined by the *bound program materialization options* and the *bound module materialization options*.

No exception is signaled when the size of the *receiver*, as specified by *bytes provided* is not large enough to hold data for all requested *bound program materialization options* and *bound module materialization options*. Instead, the data is truncated and *bytes provided* only reflects the actual amount of data returned. One of several conditions may arise, each with a different result.

If the receiver is not large enough to hold the materialization header, no data is returned for that *bound program materialization option* or *bound module materialization option*. The *offset to next entry* field in the previous materialization header, if one exists, is set to 0, and the *bytes available* field is set to reflect the amount of data actually materialized for the *bound program materialization options* or *bound module materialization options* that have already been processed. *Bytes available* will be set to 8, or *bytes provided* ; whichever is less, if the *receiver* is not big enough to hold the first materialization header.

If the *receiver* is big enough to hold the materialization header, but not big enough to hold all of the data requested by the *bound program materialization option* or *bound module materialization option*, the *partial data* flag will be set in the materialization header and as much data will be returned for which there is room. For data which consists of one continuous stream[2] (page 539) the receiver will be filled and *bytes available* will equal *bytes provided*. For data which consists of an option specific header followed by an array of homogenous elements[3] (page 539) data will be returned in such a way that no partial option specific header or array element will be returned. If there is not enough room to hold the entire option specific header, none of it will be returned. If there is room for the option specific header, but not all of the entries, only those entries that will fit will be returned. The number of entries in the option specific header will reflect the number of entries returned rather than the actual number of entires available in the module. *Bytes available* will reflect the actual amount of data returned and may not equal *bytes provided*. Note that because many option specific headers and entries are larger than the common *materialization header*, there may be more than one option for which partial data is returned.

Format of Common Materialization Header:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Offset to next entry	UBin(4)	
4	4	Bound program materialization identifier	Char(4)	
8	8	Bound module materialization identifier	Char(4)	
12	C	Bound module materialization number identifier	UBin(4)	
16	10	Flags	Char(4)	
16	10		Entry presence **0 =** No data present for entry **1 =** Data present for entry	Bit 0
16	10		Partial data	Bit 1

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = All data in materialization was returned	
			1 = Partial data was returned because receiver was too small to hold all data for the requested option	
16	10		Valid materialization data	Bit 2
			0 = The data requested in this materialization request is never present for the type of bound program being materialized.	
			1 = The data requested in this materialization request may be present for the type of bound program being materialized.	
16	10		Reserved (binary 0)	Bits 3-31
20	14	Reserved (binary 0)	Char(12)	
32	20	—- End —-		

Offset to next entry

This contains the offset from the beginning of this entry to the beginning of the next entry. It will contain zero if this is the last entry.

Bound program materialization identifier

This indicates which portion of the bound program is contained in this entry. It is the bit which was on in *bound program materialization options* that resulted in this data being materialized. Either no bits, or a single bit of this field will be on. If no bits of this field are on, then the data contained in this entry is indicated by the *bound module materialization identifier* field.

Bound module materialization identifier

This indicates which portion of the module, indicated by the *bound module materialization number identifier* field, is contained in this entry. It is the bit which was on in the *bound module materialization options* field that resulted in this data being materialized. Either a single bit or no bit of this field will be on. If no bit is on, then the data contained in this entry is indicated by the *bound program materialization identifier* field. If a bit is set on, then that type of information will be returned in the entry.

Bound module materialization number identifier

If a bit of the *bound module materialization identifier* field is on, then this is the number of the module for which information has been materialized in this entry, and this field will not be 0.

If no bits of the *bound module materialization identifier* field are on, then this field will be 0.

Flags

This field specifies information about the item being materialized.

The **entry presence** field specifies whether there is data available for the requested item. Some items may not be encapsulated into the object, so no data will be returned when their materialization is requested.

The **partial data** field specifies that only a portion of the data was returned because sufficient space was not present in the *receiver* to hold all of the data for the requested materialization option.

The **valid materialization** field specifies whether the requested information is valid to be materialized for the type of bound program that is being materialized. For example, *specific bound program information* is not valid for a bound service program. Even if data may be present for the type of bound program being materialized does not mean that it actually is. Refer to the *entry present* field to see if it is.

Format of materialized general bound program information:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Length in bytes of materialization	UBin(4)
4	4	Reserved	Char(264)
		Note: Reserved for IBM Internal Use Only. If used, unpredictable results may occur.	
268	10C	Number of secondary associated spaces	UBin(4)
272	110	Activation group attributes	Char(4)
272	110	Target activation group	Char(1)
		0 = Default activation group	
		1 = Caller's activation group	
		2 = Named activation group	
		3 = Unnamed activation group	
		4 = Named shared activation group	
		5 = Unnamed shared activation group	
		6-255 Reserved	
273	111	Reserved	Char(3)
		Note: Reserved for IBM Internal Use Only. If used, unpredictable results may occur.	
276	114	Activation group name	Char(30)
306	132	Reserved	Char(14)
		Note: Reserved for IBM Internal Use Only. If used, unpredictable results may occur.	
320	140	Coded character set identifier	UBin(2)
322	142	Composite language version	UBin(2)

Offset Dec	Hex	Field Name	Data Type and Length
		All versions are represented as 16 bit values mapped as follows. **Bits 0-3** Reserved (binary 0) **Bits 4-7** Version **Bits 8-11** Release **Bits 12-15** Modification	
324	144	Composite machine version for modules All versions are represented as 16 bit values mapped as follows. **Bits 0-3** Reserved (binary 0) **Bits 4-7** Version **Bits 8-11** Release **Bits 12-15** Modification	UBin(2)
326	146	Earliest version All versions are represented as 16 bit values mapped as follows. **Bits 0-3** Reserved (binary 0) **Bits 4-7** Version **Bits 8-11** Release **Bits 12-15** Modification	UBin(2)
328	148	Creation target version All versions are represented as 16 bit values mapped as follows. **Bits 0-3** Reserved (binary 0) **Bits 4-7** Version **Bits 8-11** Release **Bits 12-15** Modification	UBin(2)
330	14A	Version on which creation occurred All versions are represented as 16 bit values mapped as follows. **Bits 0-3** Reserved (binary 0) **Bits 4-7** Version **Bits 8-11** Release **Bits 12-15** Modification	UBin(2)
332	14C	Bound program identifier	Char(1)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
		0 = Reserved		
		1 = Bound Program		
		2 = Bound Service Program		
		3-255 = Reserved		
333	14D	Compression status	Char(1)	
333	14D		Executable portion	Bit 0
			0 = Executable portion is not compressed	
			1 = Executable portion is compressed	
333	14D		Observable portion	Bit 1
			0 = Observable portion is not compressed	
			1 = Observable portion is compressed	
			If the compression status of the observable portion of the program is uncompressed, it is not a guarantee that the observable portion exists. The observable portion status should be checked to ensure that the observable portion exists before attempting to access the observable portion of the program.	
333	14D		Reserved (binary 0)	Bits 2-7
334	14E	Composite low optimization level	UBin(2)	
336	150	Composite high optimization level	UBin(2)	
338	152	Observable portion status	Char(1)	
338	152		Extended observability storage area exists	Bit 0
			0 = Extended observability storage area does not exist	
			1 = Extended observability storage area exists	
			Note that a value of zero in this field just means that no observable data exists separate from the portion that is encapsulated in the executable portion of the program. For more information see the Observability Provided and Encapsulated Observability Attributes in the materialized general bound program information.	
338	152		Program creation data existence	Bit 1
			0 = The program creation data does not exist	
			1 = The program creation data does exist	
338	152		Module creation data existence	Bit 2
			0 = Some or all of the module creation data does not exist	
			1 = All of the module creation data does exist	
338	152		Reserved (binary 0)	Bits 3-7

Offset Dec	Hex	Field Name	Data Type and Length	
339	153	Program application profiling attributes	Char(1)	
339	153		Program procedure order profiled	Bit 0
			0 = Program is not procedure order profiled	
			1 = Program is procedure order profiled	
339	153		Program basic block reordering attempted	Bit 1
			0 = No modules bound in the program have been basic block reordered	
			1 = Basic block reordering was attempted for one or more of the modules bound in the program	
339	153		Program ready for application profiling collection	Bit 2
			0 = No modules bound in the program are hooked for application profiling	
			1 = One or more of the modules bound in the program are hooked for application profiling	
			339	
			153	
			Reserved (binary 0)	
			Bits 3-7	
			340	
			154	
			Number of modules with application profiling attributes	
			UBin(4)	
			344	
			158	
			Teraspace attributes	
			Char(1)	
			344	
			158	
			Program contains teraspace capable modules	
			Bit 0	
			0 = No modules bound in the program are teraspace capable	
			1 = One or more modules bound in the program are teraspace capable	
			344	
			158	
			Program entry procedure teraspace capable	
			Bit 1	
			0 = Program entry procedure is not teraspace capable	
			1 = Program entry procedure is teraspace capable	

344
158
All modules teraspace capable
Bit 2

0 = One or more modules in the bound program are not teraspace capable

1 = All modules in the bound program are teraspace capable

344
158
Program automatic and static storage location
Bits 3-4

00 = Automatic and static storage are allocated from single level store

01 = Automatic and static storage are allocated from teraspace

10 = Automatic and static storage are allocated from either single level store or teraspace, depending upon the activation

11 = Reserved

344
158
Reserved (binary 0)
Bits 5-7
345
159
Reserved (binary 0)
Char(167)
512
200
—- End —-

Length in bytes of materialization
This is the number of bytes materialized. For the general bound program information this will always be a constant 512.

Number of secondary associated spaces
This is the number of secondary associated spaces currently associated with the object.

Activation group attributes
The *activation group attributes* specify characteristics of the activation group into which the program will be activated.

Target activation group
This is the *target activation group* value specified when the bound program or bound service program was created.

Activation group name
This is the *activation group name* specified when the bound program or bound service program was created.

Coded character set identifier
This is the CCSID value of the bound program or bound service program.

Composite language version
This is the earliest version of the operating system on which the languages used for the bound modules will allow the bound program object to be saved. This is a composite[4] (page 539) of all of the language versions of the modules bound into this program.

Composite machine version for modules
This is the earliest version of the operating system on which all of the modules bound into the program can be re-created, assuming the required module creation templates are encapsulated in the program.

Earliest version
This is the earliest version of the operating system for which the machine will allow the bound program to be saved.

Creation target version
This is the version of the operating system for which the bound program object was created.

Version on which creation occurred
This is the version of the operating system on which the bound program object was created.

Bound program identifier
This field identifies the type of bound program being materialized.

Compression status

This field identifies whether the executable or the observable portions of the bound program or bound service program are compressed.

Composite low optimization level

This field reflects the lowest level of optimization of all the modules bound into the program. Some modules of the program may have a higher optimization level than indicated by this field.

Composite high optimization level

This field reflects the highest level of optimization of all the modules bound into the program. Some modules of the program may have a lower optimization level than indicated by this field.

Observable portion status

This field describes the status of the observability data of the program. It includes the following flag fields

Extended observability storage area exists

This field specifies whether the extended storage area for observability data exists for the program. It is an indication of whether or not any observable segments exist in the program.

Program creation data existence

This field specifies whether the program creation data that was provided when the program was created still exists in the program.

Module creation data existence

This field specifies whether all of the module creation data that was provided when the modules were created still exists in the program.

In order for a program to be eligible for retranslation, the program creation data existence bit must be set. If the module creation data existence bit is also set, this is an indication that the program has the required observability for retranslation. If the module creation data existence field is not set, the observable status of each of the modules can be found.

Program application profiling attributes

These attributes identify application profiling attributes of the program object.

Program procedure order profiled

This attribute indicates whether the bound program was created with a procedure order list. Packaging order of the procedures can affect run time performance.

Program basic block reordering attempted

This attribute indicates whether the bound program contains any modules for which basic block reordering was attempted (i.e. application profiling data that was collected was applied to one or more modules bound into this program). The packaging order of code in the procedures within the modules can affect run time performance.

Program ready for application profiling collection

This attribute indicates whether the bound program contains any modules that are hooked for application profiling collection (i.e. the modules contain hooks that enable data to be collected by application profiling).

Number of modules with application profiling attributes

This is the number of modules bound into the program that are either hooked for application profiling data collection or have been basic block reordered. To determine to which attributes this number applies, use the *program ready for application profiling collection* and the *program basic block reordering attempted* bits of the *program application profiling attributes*. They are mutually exclusive.

Teraspace attributes These attributes identify teraspace attributes of the program object.

Program contains teraspace capable modules

This attribute indicates whether or not one or more bound modules are teraspace capable. If so, then a call from a procedure in that module will only be able to pass a teraspace address to procedures in other teraspace capable programs.

Program entry procedure teraspace capable

This attribute indicates whether or not the program entry procedure was created teraspace capable. If so, it may be called with parameters stored in teraspace and after being called will be allowed to address teraspace.

All modules teraspace capable

This attribute indicates whether all modules in the bound program are teraspace capable. If so, then any procedure can be passed a teraspace address when called.

Automatic and static storage location

This attribute identifies where the automatic and static storage for the program will be allocated at run time. If the attribute has a value of single level store, the automatic and static storage will be allocated from single level store. A value of teraspace means the automatic and static storage will be in teraspace. A value of either indicates that the program is capable of running in single level store and teraspace, and the activation group into which the program is activated will determine from which type of storage the automatic and static storage are allocated.

Format of materialized program copyright strings:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Length in bytes of materialization	UBin(4)
4	4	Version of copyright creation extension	UBin(4)
8	8	Number of copyright statements in the pool	UBin(4)
12	C	Reserved	Char(4)
16	10	Copyright string pool	Char(*)
*	*	—- End —-	

Length in bytes of materialization This is the number of bytes materialized.

Version of copyright creation extension This is the version of the copyrights when the module was created.

Number of copyright strings in the pool This is the number of copyright strings that follow.

Copyright statement pool This is the data for all of the copyright strings. Each copyright string consists of a 4 byte length followed by the text of the string. The length reflects the actual length of the copyright string and does not include the length of the length field. All copyright strings along with their lengths are placed contiguously in the buffer with no intervening padding.

Format of the materialized bound service programs information:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Length in bytes of materialization	UBin(4)	
4	4	Number of service programs bound to this program	UBin(4)	
8	8	Reserved (binary 0)	Char(8)	
16	10	Array of bound service program records	[*] Char(48)	
16	10		Bound service program ID	Char(24)
16	10		Bound service program context object type	C
17	11		Bound service program context object subtype	C
18	12		Bound service program context name	C
28	1C		Bound service program object type	C
29	1D		Bound service program object subtype	C
30	1E		Bound service program name	C
40	28		Referentially bound program signature	Char(16)
56	38		Reserved (binary 0)	Char(8)
*	*	—- End —-		

Length in bytes of materialization
The number of bytes materialized. This will be 16 + (N * 48) where N is the number of bound service programs- those programs that contain exports that resolve imports in the bound program.

Number of service programs bound to this program
This is the number of bound service programs bound to the bound program.

Array of bound service program records

This array contains one record for each bound service program bound to the bound program. Each record contains the following information

Bound service program context type

This is the object type of the context with the given name.

Bound service program context subtype

This is the object subtype of the context with the given name.

Bound service program context name

This is the context specified during program creation where this bound service program was found when the bound program was created. This value could be set with all hex zeroes, in which case the name resolution list is used to locate the given bound service program. The context name or name resolution list is searched using the name space of the thread in which the activation occurs (See the RSLVSP instruction for a description of name spaces.)

Bound service program type

This is the object type of the program with the given name.

Bound service program subtype

This is the object subtype of the program with the given name.

Bound service program name

This is the name of the bound service program specified during program creation.

Bound service program signature

This is the signature of the bound service program that was used to match against the current signature of the bound program.

Format of the materialized bound modules information:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Length in bytes of materialization	UBin(4)
4	4	Number of modules bound into this program	UBin(4)
8	8	Reserved (binary 0)	Char(8)
16	10	Array of bound module records	[*] Char(80)
16	10	Bound module ID	Char(60)

Offset Dec	Hex	Field Name	Data Type and Length	
16	10		Module qualifier	Char(3
46	2E		Module name	Char(3
76	4C		Reserved (binary 0)	Char(20)
*	*	—- End —-		

Length in bytes of materialization
The number of bytes materialized. This will be 16 + (N * 80) where N is the number of modules bound into the bound program.

Number of modules bound into this program
This is the number of modules bound into the bound program.

Array of bound module records
This array contains one record for each module bound into the bound program. Each record contains the following information

Module qualifier

This is the qualifier specified during program creation where this module was found when the bound program was created. The module qualifier is used to differentiate between two different modules of the same name. This usually contains a context name.

Module name

This is the name of the module.

Format of the materialized bound program string directory component:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Length in bytes of materialization	UBin(4)	
4	4	Reserved (binary 0)	Char(12)	
16	10	String pool	Char(*)	
16	10		Length of the string	UBin(4)
20	14		CCSID of the string	UBin(2)
22	16		String	Char(*)
*	*	—- End —-		

Length in bytes of materialization
The number of bytes materialized. This will be 16 + the length of the string pool.

String pool
A memory area containing the strings defined for this program. It can be of any length addressable by a UBin(4). It contains a series of strings and lengths. String IDs specified in other materialized components can be used as indexes into this string pool.

Length of string
The length of the next string. This field contains the length of the string only, and does not include the length of the either the length or the CCSID field. The length field of a string is not subject to alignment considerations.

CCSID of string
The character code set identifier of this string. This string is encoded in the given CCSID, which is the CCSID of the module object from which this string is originally declared. The CCSID field of a string is not subject to alignment considerations.

String
Character buffer which contains one string. Its length is defined by the length field.

Format of the materialized bound program limits:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Length in bytes of materialization	UBin(4)
4	4	Reserved (binary 0)	Char(12)
16	10	Current size of bound program	UBin(4)
20	14	Maximum number of associated spaces	UBin(4)
24	18	Current number of associated spaces	UBin(4)
28	1C	Maximum number of modules bindable into program	UBin(4)
32	20	Current number of modules bound into program	UBin(4)
36	24	Maximum number of service programs bindable to program	UBin(4)
40	28	Current number of service programs bound to program	UBin(4)
44	2C	Maximum size of bound program string directory	UBin(4)
48	30	Current size of bound program string directory	UBin(4)
52	34	Maximum size of bound program copyright strings	UBin(4)
56	38	Current size of bound program copyright strings	UBin(4)
60	3C	Maximum number of auxiliary storage segments	UBin(4)
64	40	Current number of auxiliary storage segments	UBin(4)
68	44	Maximum number of static storage frames	UBin(4)
72	48	Current number of static storage frames	UBin(4)
76	4C	Maximum number of program procedure exports	UBin(4)
80	50	Current number of program procedure exports	UBin(4)
84	54	Maximum number of program data exports	UBin(4)
88	58	Current number of program data exports	UBin(4)
92	5C	Maximum number of signatures	UBin(4)
96	60	Current number of signatures	UBin(4)
100	64	Minimum amount of static storage required	UBin(4)
104	68	Maximum amount of static storage required	UBin(4)
108	6C	Reserved (binary 0)	Char(4)
112	70	Eight byte version of minimum amount of static storage required	UBin(8)
120	78	Eight byte version of maximum amount of static storage required	UBin(8)
128	80	Reserved (binary 0)	Char(128)
256	100	—- End —-	

Length in bytes of materialization
The number of bytes materialized. This will always be a constant 256.

Current size of bound program
This is the current size, in machine-dependent units, of the bound program being materialized.

Maximum number of associated spaces
This is the maximum number of associated spaces allowed for the bound program being materialized.

Current number of associated spaces
This is the current number of associated spaces allocated to the bound program being materialized.

Maximum number of modules bindable into program
This is the maximum number of modules that can be bound into a bound program.

Current number of modules bound into program
This is the current number of modules bound into the bound program being materialized.

Maximum number of service programs bindable to program
This is the maximum number of bound service programs that can be bound to a bound program. These bound service programs contain exports to which imports from a bound program resolve.

Current number of service programs bound to program
This is the current number of bound service programs bound to the bound program being materialized.

Maximum size of bound program string directory
This is the maximum size, in bytes, of the bound program string directory.

Current size of bound program string directory
This is the current size, in bytes, of the bound program string directory.

Maximum size of bound program copyright strings
This is the maximum size, in bytes, of the bound program copyright strings.

Current size of bound program copyright strings
This is the current size, in bytes, of the bound program copyright strings.

Maximum number of auxiliary storage segments
This is the maximum number of auxiliary storage segments allowed for a bound program.

Current number of auxiliary storage segments
This is the current number of auxiliary storage segments in the bound program being materialized.

Maximum number of static storage frames
This is the maximum number of static storage frames allowed for a bound program.

Current number of static storage frames
This is the current number of static storage frames required by the bound program being materialized.

Maximum number of procedure exports
This is the maximum number of procedures that are allowed to be exported from a bound program. If the bound program being materialized is not a bound service program, then this value will be zero.

Current number of procedure exports
This is the current number of procedures exported from the bound program being materialized. If the bound program being materialized is not a bound service program, then this value will be zero.

Maximum number of data exports
This is the maximum number of data items that are allowed to be exported from a bound program. If the bound program being materialized is not a bound service program, then this value will be zero.

Current number of data exports
This is the current number of data items exported from the bound program being materialized. If the bound program being materialized is not a bound service program, then this value will be zero.

Maximum number of signatures
This is the maximum number of signatures allowed for a bound program. If the bound program being materialized is not a bound service program, then this value will be zero.

Current number of signatures
This is the current number of signatures contained in the bound program being materialized. If the bound program being materialized is not a bound service program, then this value will be zero.

Minimum amount of static storage required.
This is the smallest amount of static storage that is required for the bound program or service program. This measure is in bytes. The actual amount of static storage that is used may be anywhere between the minimum and the maximum amounts of required static storage, inclusive. If the size is 4 gigabytes (4,294,967,296) or more, a value of 4,294,967,295 will be returned in the field. In this case, the *eight byte version of minimum amount of static storage required* field should be used to get the size in bytes.

Maximum amount of static storage required.
This is the largest amount of static storage that may be required for the bound program or service program. This measure is in bytes. The actual amount of static storage that is used may be anywhere between the minimum and the maximum amounts of required static storage, inclusive. If the size is 4 gigabytes (4,294,967,296) or more, a value of 4,294,967,295 will be returned in the field. In this case, the *eight byte version of maximum amount of static storage required* field should be used to get the size in bytes.

Eight byte version of minimum amount of static storage required
This is the smallest amount of static storage that is required for the bound program or service program. This measure is in bytes. The actual amount of static storage that is used may be anywhere between the minimum and the maximum amounts of required static storage, inclusive.

Eight byte version of maximum amount of static storage required

This is the largest amount of static storage that may be required for the bound program or service program. This measure is in bytes. The actual amount of static storage that is used may be anywhere between the minimum and the maximum amounts of required static storage, inclusive.

Format of the materialized activation group data imports:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Length in bytes of materialization	UBin(4)	
4	4	Number of activation group data imports	UBin(4)	
8	8	Reserved (binary 0)	Char(8)	
16	10	Array of activation group data imports	[*] Char(16)	
16	10		String ID	UBin(4)
20	14		Reserved (binary 0)	Char(12)
*	*	—- End —-		

Length in bytes of materialization

The number of bytes materialized. This will be (N+1)*16, where N is the number of activation group data imports contained in the bound program or bound service program.

Number of activation group data imports

The number of activation group data imports contained in the bound program or bound service program.

Array of activation group data imports

This array contains one record for each data item contained in the program or bound service program. Each record contains the following information:

String ID

This is the identification used to extract the name of this data item from the program string directory.

Format of the materialized activation group data exports:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Length in bytes of materialization	UBin(4)	
4	4	Number of activation group data exports	UBin(4)	
8	8	Reserved (binary 0)	Char(8)	
16	10	Array of activation group data exports	[*] Char(16)	
16	10		String ID	UBin(4)
20	14		Strength of data item 0 = Reserved 1 = Export Strongly 2 = Export Weakly 3-255 = Reserved	Char(1)
21	15		Reserved (binary 0)	Char(3)
24	18		Length of data item	UBin(4)
28	1C		Reserved (binary 0)	Char(4)
*	*	—- End —-		

Length in bytes of materialization
The number of bytes materialized. This will be (N+1)*16, where N is the number of activation group data exports contained in the bound program or bound service program.

Number of activation group data exports
The number of activation group data exports contained in the bound program or bound service program.

Array of activation group data exports
This array contains one record for each data item contained in the program or bound service program. Each record contains the following information:

String ID
This is the identification used to extract the name of this data item from the program string directory.

Strength of data item
This field indicates whether the activation group export is exported strongly or weakly.

Length of data item
The size in bytes of the activation group export.

Format of the materialized specific bound program information: Specific bound program information can only be materialized for bound programs, and not for bound service programs.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Length in bytes of materialization	UBin(4)	
4	4	Reserved (binary 0)	Char(12)	
16	10	Program entry procedure information	Char(16)	
16	10		Module number containing program entry procedure	UBin(4)
20	14		Program entry procedure string ID	UBin(4)
24	18		Minimum parameters	UBin(2)
26	1A		Maximum parameters	UBin(2)
28	1C		Reserved (binary 0)	Char(4)
32	20	Reserved (binary 0)	Char(32)	
64	40	—- End —-		

Length in bytes of materialization
The number of bytes materialized. This will always be a constant 64.

Module number containing program entry procedure
This is the number, in the bound modules information, of the module which contains the program entry procedure for this bound program.

Program entry procedure string ID
This is the string ID for the name of this program entry procedure.

Minimum parameters
This is the minimum number of parameters that the program entry procedure can accept.

Maximum parameters
This is the maximum number of parameters that the program entry procedure can accept.

Format of the materialized signatures information: Signatures information can only be materialized for bound service programs, and not for bound programs.

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Length in bytes of materialization	UBin(4)		
4	4	Number of signatures contained in the program	UBin(4)		
8	8	Reserved (binary 0)	Char(8)		
16	10	Array of signatures	[*] Char(16)		
16	10		Signature		Char(16)
*	*	—- End —-			

Length in bytes of materialization
The number of bytes materialized. This will be (N+1)*16, where N is the number of signatures contained in the program.

Number of signatures contained in the program
This is the number of signatures contained in the program.

Array of signatures
This array contains one record for each signature contained in the program. Each record contains the following information. The first record contains the current signature.

Signature
A signature of the service program.

Format of the materialized exported program procedure information: Exported program procedure information can only be materialized for bound service programs, and not for bound programs.

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Length in bytes of materialization	UBin(4)		
4	4	Number of exported procedures	UBin(4)		
8	8	Reserved (binary 0)	Char(8)		
16	10	Array of program exports	[*] Char(16)		
16	10		String ID for procedure export		UBin(4)
20	14		Export number		UBin(4)
24	18		Procedure parameter mask		Char(2)
26	1A		Reserved (binary 0)		Char(6)
*	*	—- End —-			

Length in bytes of materialization
The number of bytes materialized. This will be (N+1)*16, where N is the number of exported procedures.

Number of exported procedures
This is the number of procedures exported from the service program.

Array of program exports

This array contains one record for each procedure exported from the service program. Each record contains the following information:

String ID for procedure export

This is the identification used to extract the name of this exported procedure from the program string directory.

Export number

This is the number of this exported procedure.

Procedure parameter mask

The *procedure parameter mask* indicates the parameter characteristics of the procedure. The procedure parameter mask will be binary zero if the program using this instruction is executing in user-state.

Format of the materialized exported program data information: Exported program data information can only be materialized for bound service programs, and not for bound programs.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Length in bytes of materialization	UBin(4)	
4	4	Number of exported data items	UBin(4)	
8	8	Reserved (binary 0)	Char(8)	
16	10	Array of data exports	[*] Char(16)	
16	10		String ID for data export	UBin(4)
20	14		Export number	UBin(4)
24	18		Data item size	UBin(4)
28	1C		Reserved (binary 0)	Char(4)
*	*	—- End —-		

Length in bytes of materialization

The number of bytes materialized. This will be (N+1)*16, where N is the number of exported data items.

Number of exported data items

This is the number of data items exported from the service program.

Array of data exports

This array contains one record for each data item exported from the service program. Each record contains the following information:

String ID for data export

This is the identification used to extract the name of this exported data item from the program string directory.

Export number

This is the number of this exported data item.

Data item size

This is the size in bytes of the exported data item

Format of materialized general module information:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Length in bytes of materialization	UBin(4)	
4	4	Reserved	Char(12)	
16	10	Reserved **Note:** Reserved for IBM Internal Use Only. If used, unpredictable results may occur.	Char(276)	
292	124	Coded character set identifier	UBin(2)	
294	126	Data required for machine retranslation	Char(1)	
294	126		All data required for machine retranslation is present **0 =** No **1 =** Yes	Bit 0
294	126		Reserved (binary 0)	Bits 1-7
295	127	Reserved	Char(9)	
304	130	Creation target version All versions are represented as 16 bit values mapped as follows. **Bits 0-3** Reserved (0) **Bits 4-7** Version **Bits 8-11** Release **Bits 12-15** Modification	Char(2)	
306	132	Language version	Char(2)	

Offset Dec	Offset Hex	Field Name	Data Type and Length
		All versions are represented as 16 bit values mapped as follows.	
		Bits 0-3 Reserved (0)	
		Bits 4-7 Version	
		Bits 8-11 Release	
		Bits 12-15 Modification	
308	134	Version on which creation occurred	Char(2)
		All versions are represented as 16 bit values mapped as follows.	
		Bits 0-3 Reserved (0)	
		Bits 4-7 Version	
		Bits 8-11 Release	
		Bits 12-15 Modification	
310	136	Earliest version	Char(2)
		All versions are represented as 16 bit values mapped as follows.	
		Bits 0-3 Reserved (0)	
		Bits 4-7 Version	
		Bits 8-11 Release	
		Bits 12-15 Modification	
312	138	Reserved	Char(16)
328	148	Number of secondary associated spaces	UBin(4)
332	14C	Reserved	Char(16)
348	15C	Reserved	Char(2)
		Note: Reserved for IBM Internal Use Only. If used, unpredictable results may occur.	
350	15E	Module state	UBin(2)
		Hex 0001 = User state	
		Hex 8000 = System state	
		Hex 0000 = Inherit state	
352	160	Compiler name	Char(20)
372	174	Program entry procedure	Char(16)
372	174	Program entry procedure attributes	Char(4)
372	174	Program entry procedure exists	Bit 0
		0 = Program entry procedure does not exist in this module	
		1 = Program entry procedure exists in this module	
372	174	Reserved (binary 0)	Bits 1-31
376	178	Program entry procedure dictionary ID	UBin(4)

Offset Dec	Hex	Field Name	Data Type and Length	
380	17C		Program entry procedure string ID	UBin(4)
384	180		Program entry procedure minimum parms	UBin(2)
386	182		Program entry procedure maximum parms	UBin(2)
388	184	Module application profiling attributes	Char(1)	
388	184		Hooks for application profiling are present	Bit 0
			0 = Hooks are not present	
			1 = Hooks are present	
388	184		Basic block reordering attempted	Bit 1
			0 = Basic block reordering on the procedures in this module has not been attempted.	
			1 = Basic block reordering on the procedures in this module has been attempted. The *procedure basic block attributes* under the *procedure definitions* further identifies which procedures have been basic block reordered.	
388	184		Reserved (binary 0)	Bits 2-7
389	185	Module teraspace attributes	Char(1)	
389	185		Module teraspace capable	Bit 0
			0 = Module is not teraspace capable	
			1 = Module is teraspace capable	
389	185		Module automatic and static storage location	Bits 1-2
			00 = Automatic and static storage are allocated from single level store	
			01 = Automatic and static storage are allocated from teraspace	
			10 = Automatic and static storage are allocated from either single level store or teraspace, depending upon the activation group into which the program is activated.	
			11 = Reserved	
389	185		Reserved (binary 0)	Bits 3-7
390	186	Reserved	Char(122)	
512	200	—- End —-		

Length in bytes of materialization
This is the number of bytes materialized. For the general module information this will always be a constant 512.

Coded character set identifier
The CCSID defines the code page of the symbols in the string directory.

Creation target version
This is the version of the operating system for which the module object was created.

Data required for machine retranslation
This indicates whether the data required for machine retranslation is present. This data can be present even if the *encapsulated observability attributes* indicated the component is not present.

Language version
This is the earliest version of the operating system on which language used will allow the module object to be saved.

Version on which creation occurred
This is the version of the operating system that was running on the system where the module object was created.

Earliest version
This is the earliest version of the operating system for which the machine will allow the module object to be saved.

Number of secondary associated spaces
This is the number of secondary associated spaces currently associated with the object.

Module state
This is the state of the module object.

Compiler name
This identifies the compiler which translated the user's source language.

Program entry procedure
This identifies the program entry procedure if one is present in the module.

Program entry procedure attributes

This bit mapped field identifies attributes of the program entry procedure.

The *program entry procedure existence* field specifies whether a program entry procedure is present in the module being materialized.

Program entry procedure dictionary ID

The dictionary ID is used as a handle to uniquely identify the procedure.

Program entry procedure string ID

The string ID may be used to extract the character string which is the procedure name from the string pool.

Program entry procedure minimum parms

This is the minimum number of parameters allowed by the program entry procedure.

Program entry procedure maximum parms

This is the maximum number of parameters allowed by the program entry procedure.

Module application profiling attributes
This bit mapped field identifies the application profiling attributes of the module.

The **hooks for application profiling are present** field specifies whether the module is ready for application profiling data collection when it is bound into a program.

The **basic block reordering attempted** field specifies whether reordering of the basic blocks within the procedures of this module has been attempted. To determine which procedures have basic blocks reordered see the *procedure basic block attributes* under the *procedure definitions*.

Module teraspace attributes

This bit mapped field identifies the teraspace attributes of the module.

The **module teraspace capable** field specifies whether or not the module is allowed to use teraspace at run time.

The **module automatic and static storage location** field indicates where the automatic and static storage for this module will be allocated at run time, when the module is bound into a program. The possible values are single level store, teraspace, or either single level store or teraspace, depending upon the activation group into which the program is activated.

Format of the materialized module string directory component:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Length in bytes of materialization	UBin(4)	
4	4	Reserved (binary 0)	Char(12)	
16	10	String pool	Char(*)	
16	10		Length of the string	UBin(4)
20	14		String	Char(*)
*	*	—- End —-		

Length in bytes of materialization

The number of bytes materialized. This will be 16 + the length of the string pool.

String pool

A memory area containing the strings defined for this module. It can be of any length addressable by a UBin(4). It contains a series of strings and lengths. String IDs specified in other materialized components can be used as indexes into this string pool.

Length of string

The length of the next string. This field contains the length of the string only, and does not include the length of the length field, itself. The length field of a string is not subject to alignment considerations.

String

Character buffer which contains one string. Its length is defined by the length field.

Format of the materialized module copyright strings: The format of the materialized module copyright strings is the same as for the materialized program copyright strings.

Template Value Invalid exception reason codes: This instruction supports setting of the optional reason code field in the exception data which can be retrieved when the template value invalid exception is signaled. When the first byte of the reason code is not zero, the exception is being signaled because one of the materialization receivers is not valid.

00

Bound Program Materialization Template (pointed to by operand 1 of this instruction)

01 Size of template is not sufficient to hold number of requests specified.

0n	nth materialization request is not valid. **01** The *receiver* is not aligned on a 16 byte boundary. **02** The materialization request *bytes provided* is less than 8. **03** The materialization request contains no *materialization options* or invalid *materialization options*. If the *length of field* data is 8, then no materialization options were specified and the *offset in field in bits* data will be 0. Otherwise, an invalid option was specified and the provided *offset to field in bytes* and *offset in field in bits* data will identify the invalid materialization option. **04** The materialization request contains a *module materialization number* that is greater than the number of modules bound into the program. **05** The materialization request contains a non-zero *module materialization number*, but no *module materialization options*. **06** The materialization request contains a non-zero reserved field.

Warning: Temporary Level 3 Header

Authorization Required

•

- Retrieve
  –
  – Operand 2
- Execute
  –
  – Contexts referenced for address resolution
  – Bound service programs required to activate the program referenced by operand 2

Lock Enforcement

•

- Materialize
  –
  – Operand 2
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2204 Object Not Eligible for Operation

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220A Program Not Eligible for Operation

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3802 Template Size Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Footnotes:

[1] Referential extensions are data streams that are not included in the creation templates, but are pointed to by a space pointer in the template. This also includes the module creation template extension.

[2] The items that fall into this category are general bound program information, bound program limits, specific bound program information, specific bound service program information, general module information, bound program string directory component, module string directory component and module copyright strings.

[3] The items which fall into this category are bound service programs information, bound modules information, signatures information, program copyright strings, exported program procedure information, activation group data imports, activation group data exports, and exported program data information.

[4] A composite version is defined to be the latest version, in time, of all of the versions comprising the composite. Given back-level compatibility, this would be the earliest version of the operating system on which all of the comprising versions would be compatible.

Materialize Context (MATCTX)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0133	Receiver	Context	Materialization options

Operand 1: Space pointer.

Operand 2: System pointer or null.

Operand 3: Character scalar.

Bound program access
Built-in number for MATCTX is 57. MATCTX (receiver : address context : address of system pointer OR null operand materialization_options : address)

Description: Based on the contents of the *materialization options* specified by operand 3, the symbolic identification and/or system pointers to all, or a selected set, of the objects addressed by the *context* specified by operand 2 are materialized into the *receiver* specified by operand 1.

If operand 2 is a null operand and the *machine context selection* field is zero, then the machine context for the system ASP is materialized. If operand 2 is a null operand and the *machine context selection* field is one, then the independent ASP machine context for the *independent ASP number* specified is materialized.

Usage note:

To get a list of every context on a system, first use the MATRMD instruction to retrieve a list of every independent ASP (and the system ASP) on the system, using *selection option* hex 1F. For every independent ASP in the list, the field *independent ASP* will be set to binary 1. For each independent ASP (and the system ASP), the *ASP number* will be materialized. Each of the machine contexts can then be materialized using the MATCTX instruction. For each call to this instruction, the *independent ASP number* should be set to one of the *ASP number* values materialized from MATRMD. The *machine context selection* field should be set to binary 1, the *object ID selection* field should be set to hex 1, and the *type code* should be set to hex 04.

The *materialization options* operand has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Materialization control	Char(2)
0	0		Information requirements (1 = materialize)
0	0		Reserved (binary 0)
0	0		Extended context attributes
0	0		Validation
			0 = Validate system pointers
			1 = No validation
0	0		System pointers
0	0		Symbolic identification
1	1		Selection criteria
1	1		Reserved (binary 0)
1	1		Materialize hidden contexts
			0 = Do not materialize hidden contexts within the op
			1 = Materialize all contexts including hidden contexts
1	1		Machine context selection
			0 = Materialize the system machine context.
			1 = Materialize the independent ASP machine contex
1	1		Modification date/time selection

Offset Dec	Offset Hex	Field Name	Data Type and Length
			0 = Do not select by *modification date/time*
			1 = Select by *modification date/time*
1	1		Object ID selection
			Hex 0 = All entries
			Hex 1 = Type code selection
			Hex 2 = Type code/subtype code selection
			Hex 4 = Name selection
			Hex 5 = Type code/name selection
			Hex 6 = Type code/subtype code/name selection
			Hex E = Context entries collating at and above the spe
2	2	Length of name to be used for search argument	Bin(2)
4	4	Type code	Char(1)
5	5	Subtype code	Char(1)
6	6	Name	Char(30)
36	24	Timestamp	Char(8)
44	2C	Independent ASP number	Char(2)
46	2E	—- End —-	

The materialization control **information requirements** field in the *materialization options* operand specifies the information to be materialized for each selected entry. Symbolic identification and system pointers identifying objects addressed by the context can be materialized based on the bit setting of this field.

If the *information requirements* field is binary 0, the context attributes are materialized with no context entries. In this case, the *modification date/time selection* and *object ID selection* fields are ignored.

If the *information requirements* field is set to just return the **extended context attributes**, the context attributes and extended attributes are materialized with no context entries. In this case, the *modification date/time selection* and *object ID selection* fields are ignored.

If the **system pointers** field is set to binary 1, then the system pointers to objects in the context are returned in the *object pointer* field. If the **symbolic identification** field is set to binary 1, then the information in the *object identification* field is materialized for objects in the context.

If the **validation** attribute indicates *no validation* is to be performed, no object validation occurs and a significant performance improvement results.

When *no validation* occurs, some of the following pointers may be erroneous:

-
- Pointers to destroyed objects
- Pointers to objects that are no longer in the context
- Multiple pointers to the same object

If the **materialize hidden contexts** field is binary 1, then hidden contexts are materialized into the *receiver* (in addition to non-hidden contexts and other object types). A hidden context is denoted by the *hidden* attribute of a context. Since contexts can only be contained in a machine context, the *materialize hidden contexts* option is only useful when a machine context is being materialized. This option can only be specified when the thread is in system state. Otherwise a *template value invalid* (hex 3801) exception will be signalled.

The materialization control *modification date/time selection* and *object ID selection* fields specify the context entries from which information is to be presented. The **type code**, **subtype code**, and **name** fields contain the selection criteria when a selective materialization is specified.

When *type code* or *type/subtype codes* are part of the selection criteria, only entries that have the specified codes are considered. When a **name** is specified as part of the selection criteria, the N characters in the search criteria are compared against the N characters of the context entry, where N is defined by the **length of name to be used for search argument** field in the materialization options. The remaining characters (if any) in the context entry are not used in the comparison.

If *modification date/time selection* is 0 and *object ID selection* is 0 and the number of bytes provided in the *receiver* does not allow for materialization of at least one context entry, requests that as much of the context attributes as will fit be materialized into the *receiver* and that an estimate of the byte size correlating to the full list of context entries currently in the context be set into the *number of bytes available for materialization* field of the *receiver*. This capability of requesting an estimate of the size of a full materialization of the context provides a low overhead way of getting a close approximation of the amount of space that will be needed for an actual materialize of all context entries. This approximation may be either high or low by a few entries due to abnormal system terminations.

If the number of bytes provided in the *receiver* allows for materialization of at least one context entry, the *number of bytes available for materialization* field is set with the byte size correlating to the full list of context entries that matched the selection criteria whether or not the *receiver* provided enough room for the full list to be materialized. If **object ID selection** has a value from hex 1 through hex 6, as many context entries as will fit which match the associated *type code/subtype code/name* criteria are materialized into the *receiver*. *Object ID selection* value hex E requests that as many context entries as will fit which collate at or higher (are equal to or greater) than the specified *type code/subtype code/name* criteria be materialized into the *receiver*.

When operand 2 is a null operand, the **machine context selection** field is used to select which machine context to materialize. If 0 is specified for the *machine context selection* field, the system machine context is materialized. If 1 is specified for the *machine context selection* field, the independent ASP machine context specified by the **independent ASP number** field is materialized. If the value specified for the *independent ASP number* is not valid then a *template value invalid* (hex 3801) exception is signaled. If the associated independent ASP for the *independent ASP number* is not varied on, then a *object not available* (hex 220B) exception is signaled. If operand 2 is not a null operand, then both the *machine context selection* field and the *independent ASP number* field must be zero else a *template value invalid* (hex 3801) exception is signaled.

If **modification date/time** selection is specified, then entries are selected according to the time of last modification in addition to any object identification selection specified. The **timestamp** in the materialization control is used to determine which entries will be selected. Entries with modification timestamps greater than or equal to the *timestamp* specified in the control will be selected. Besides the additional selection done as above, the materialize will work the same as specified in the other controls.

Programming note: If the specified timestamp is for a date/time earlier than the date/time currently associated with the changed object list, all objects in the context will be inspected for their modification date. This may degrade system performance.

The format of the materialization (operand 1) is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Context identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Context options	Char(4)	
40	28		Existence attributes **0 =** Temporary **1 =** Permanent	Bit 0
40	28		Space attribute **0 =** Fixed-length **1 =** Variable-length	Bit 1
40	28		Reserved (binary 0)	Bit 2
40	28		Access group **0 =** Not a member of access group **1 =** Member of access group	Bit 3
40	28		Reserved (binary 0)	Bits 4-31
44	2C	Recovery options	Char(4)	
44	2C		Automatic damaged context rebuild option **0 =** Do not rebuild at IMPL **1 =** Rebuild at IMPL	Bit 0
44	2C		Reserved (binary 0)	Bits 1-31
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
53	35		Space alignment **0 =** The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. **1 =** The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space. Ignore the value of this field when the *machine chooses space alignment* field has a value of 1.	Bit 0
53	35		Reserved (binary 0)	Bits 1-2
53	35		Machine chooses space alignment	Bit 3

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = The space alignment indicated by the *space alignment* field is in effect.	
			1 = The machine chose the space alignment most beneficial to performance, which may have reduced maximum space capacity. The alignment chosen is a multiple of 512. Ignore the value of the *space alignment* field.	
53	35		Reserved (binary 0)	Bit 4
53	35		Main storage pool selection	Bit 5
			0 = Process default main storage pool is used for object.	
			1 = Machine default main storage pool is used for object.	
53	35		Reserved (binary 0)	Bit 6
53	35		Block transfer on implicit access state modification	Bit 7
			0 = Transfer the minimum storage transfer size for this object.	
			1 = Transfer the machine default storage transfer size for this object.	
53	35		Reserved (binary 0)	Bits 8-31
57	39	Reserved (binary 0)	Char(7)	
64	40	Reserved (binary 0)	Char(16)	
80	50	Access group	System pointer	
96	60	Extended context attributes (if requested)	Char(1)	
96	60		Changed object list	Bit 0
			0 = A changed object list does not exist	
			1 = A changed object list does exist	
96	60		Useable changed object list	Bit 1
			0 = Changed object list is in a useable state	
			1 = Changed object list is not in a useable state	
96	60		Protected	Bit 2
			0 = The context is not protected from changes	
			1 = The context is protected from changes	
96	60		Hidden	Bit 3
			0 = The context is visible	
			1 = The context is hidden	
96	60		Reserved (binary 0)	Bits 4-7
97	61	Reserved (binary 0)	Char(7)	
104	68	Current timestamp	Char(8)	
112	70	Context entry	[*] Char(16-48)	
		(repeated for each selected entry)		
112	70		Object identification (if requested)	Char(32)

Offset Dec	Hex	Field Name	Data Type and Length	
112	70		Type code	Char(1
113	71		Subtype code	Char(1
114	72		Name	Char(3
144	90		Object pointer (if requested)	System pointer
*	*	—- End —-		

The first 4 bytes of the materialization output identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled. The instruction materializes as many bytes and pointers as can be contained in the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested for materialization, the excess bytes are unchanged. No exceptions are signaled in the event that the *receiver* contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception described above.

When the **protected** attribute is set to binary 1, any attempt to insert an entry into the context or to change an entry in the context signals a *cannot change contents of protected context* (hex 4403) exception. Deleting an entry from the context will not cause this exception to be signaled. The **hidden** attribute determines whether or not this context is addressable. If a context is hidden, it affects the way the RSLVSP and MATCTX instructions behave. RSLVSP will not address hidden contexts. When MATCTX is used to materialize a machine context, hidden contexts in that machine context will only be materialized when the *materialize hidden contexts* option is set to binary 1.

The **context entry object identification** information, if requested by the *materialization options* field, is present for each entry in the context that satisfies the search criteria. If both *system pointers* and *symbolic identification* are requested by the *materialization options* field, the *system pointer* immediately follows the *object identification* for each entry.

The order of the materialization of a context is by object type code, object subtype code, and object name, all in ascending sequence.

Authorization Required

•

- Retrieve
  –
  – Operand 2
  – Device description for the specified *Independent ASP number* (when the field *Machine context selection* is set to binary 1).

Lock Enforcement

•

- Materialization
  –
  – Operand 2

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2C Program Execution

2C07 Instruction Termination

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Data Space Record Locks (MATDRECL)

Op Code (Hex)	Operand 1	Operand 2
032E	Receiver	Record selection template

Operand 1: Space pointer.

Operand 2: Space pointer.

Bound program access

```
Built-in number for MATDRECL is 49.
MATDRECL (
          receiver                  : address
          record_selection_template   : address
)
```

Description: The locks currently allocated on the specified data space record are materialized.

The current lock status of the data space record identified by the template in operand 2 is materialized into the space identified by operand 1.

The *record selection template* identified by operand 2 must be 16-byte aligned. The format of the *record selection template* is as follows.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Record selection	Char(24)	
0	0		Data space identification	System pointer
16	10		Record number	UBin(4)
20	14		Reserved	Char(4)
24	18	Lock selection	Char(1)	
24	18		Materialize data space locks held	Bit 0
			0 = Do not materialize	
			1 = Materialize	
24	18		Materialize data space locks waited for	Bit 1
			0 = Do not materialize	
			1 = Materialize	
24	18		Reserved	Bits 2-7
25	19	Template options	Char(1)	
25	19		Format for number of locks	Bit 0
			1 = Use Bin(4) for number of locks	
			0 = Use UBin(2) for number of locks	
25	19		Reserved	Bits 1-7
26	1A	Reserved	Char(6)	
32	20	—- End —-		

The **data space identification** must be a system pointer to a data space.

The **record number** is a relative record number within that data space. If the *record number* is zero then all locks on the specified data space will be materialized. If the *record number* is not valid for the specified data space a *template value invalid* (hex 3801) exception is signaled.

Both of the fields specified under **lock selection** are bits which determine the locks to be materialized. If the **materialize data space locks held** is *materialize*, the current holders of the specified data space record lock are materialized. If the **materialize data space locks waited for** is *materialize*, process information is materialized for any thread contained in the process that is waiting to lock the specified data space record.

The **format for number of locks** bit determines the format of the number of locks held and Number of locks waited for fields in the materialization template. If the bit is set to binary 1 then Bin(4) counts are used, else Bin(2) counts are used.

The materialization template identified by operand 1 must be 16-byte aligned. The format of the materialization is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Materialization data (2 possible formats)	Char(8)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			If *format for number of locks* bit=1	
8	8		Number of lock held descriptions	Bin(4)
12	C		Number of lock waited for descriptions	Bin(4)
16	10		—- End of bit=1 —-	
			If *format for number of locks* bit=0	
8	8		Number of lock held descriptions	UBin(2)
10	A		Number of lock waited for descriptions	UBin(2)
12	C		Reserved	Char(4)
16	10		—- End of bit=0 —-	
16	10	Lock held descriptions	[*] Char(32)	
		(repeated *number of lock held descriptions* times)		
16	10		Lock holder	System pointer
32	20		Record number	UBin(4)
36	24		Lock state	Char(1)
			Hex 30 = DLWK (Database lock weak) lock state	
			Hex C0 = DLRD (Database lock read) lock state	
			Hex F8 = DLUP (Database lock update) lock state	
			All other values are reserved.	
37	25		Lock holder information	Char(1)
37	25		Lock scope object type	Bit 0
			0 = Process control space	
			1 = Transaction control structure	
37	25		Lock scope	Bit 1
			0 = Lock is scoped to the *lock scope object type*	
			1 = Lock is scoped to the thread	
37	25		Reserved	Bits 2-7
38	26		Reserved (binary 0)	Char(2)
40	28		Thread ID	Char(8)
*	*	Lock waited for descriptions	[*] Char(32)	
		(repeated *number of lock waited for descriptions* times)		
*	*		Process control space	System pointer
*	*		Record number	UBin(4)
*	*		Lock state requested	Char(1)

Offset Dec	Offset Hex	Field Name	Data Type and Length
			Hex 30 = DLWK (Database lock weak) lock state
			Hex C0 = DLRD (Database lock read) lock state
			Hex F8 = DLUP (Database lock update) lock state
			All other values are reserved.
*	*		Lock waiter information — Char(1)
*	*		Lock scope object type — Bit 0 **0 =** Process control space **1 =** Transaction control structure
*	*		Lock scope — Bit 1 **0 =** Lock is scoped to the *lock scope object type* **1 =** Lock is scoped to the thread
*	*		Reserved — Bits 2-7
*	*		Reserved — Char(2)
*	*		Thread ID — Char(8)
*	*	—- End —-	

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, the excess bytes are unchanged. No exceptions are signaled in the event that the *receiver* contains insufficient area for the materialization, other than the materialization length exception described previously.

The **number of lock held descriptions** contains the number of locks held. A system pointer to the **lock holder** (the object that holds the lock), the relative **record number** which is locked, and the **lock state** are materialized in the area identified as **lock held descriptions**. The *lock holder* can be either a process control space (PCS) or a transaction control structure. The object type is determined by the value of the **lock scope object type** field. When **lock scope** has a value of *lock is scoped to the thread*, the **thread ID** field identifies the thread that holds the lock. Otherwise it is set to binary 0. These fields contain data only if *materialize data space locks held* is *materialize*.

The **number of lock waited for descriptions** contains the number of locks being waited for. A system pointer to the **process control space** (PCS) for each thread waiting for a lock, the relative **record number**, and the **lock state** which the thread is waiting for are materialized in the area identified as **lock waited for descriptions**. The *process control space* and the **thread ID** fields will identify the thread that is waiting for the lock, regardless of the **lock scope** value. These fields contain data only if *materialize data space locks waited for* is *materialize*.

A database weak record lock is only acquired thread-scoped and it only conflicts with update record locks which are thread-scoped to a different thread. The weak record lock does not conflict in any other situation.

If UBin(2) fields are requested for the *number of lock held descriptions* and *number of lock waited for descriptions*, then the maximum number that can be returned in each count is 32,767. If the actual number is greater than 32,767 for a number then that number will be set to 32,767, only the first 32,767 locks will be materialized and no exception will be signaled.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Dump Space (MATDMPS)

Op Code (Hex)	Operand 1	Operand 2
04DA	Receiver	Dump space

Operand 1: Space pointer.

Operand 2: System pointer.

Bound program access
Built-in number for MATDMPS is 83. MATDMPS (receiver : address dump_space : address of system pointer)

Description: The current attributes of the *dump space* specified by operand 2 are materialized into the *receiver* specified by operand 1.

The template identified by operand 1 must be 16-byte aligned in the space. The format of the materialization is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization (always 128 for this instruction)	Bin(4)
8	8	Object identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Object creation options	Char(4)	
40	28		Existence attributes **0 =** Temporary **1 =** Permanent	Bit 0
40	28		Space attribute **0 =** Fixed length **1 =** Variable length	Bit 1
40	28		Context **0 =** Addressability not in context **1 =** Addressability in context	Bit 2
40	28		Reserved (binary 0)	Bits 3-12
40	28		Initialize space	Bit 13
40	28		Reserved (binary 0)	Bits 14-31
44	2C	Recovery options	Char(4)	
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
57	39	Reserved	Char(7)	
64	40	Context	System pointer	
80	50	Reserved	Char(16)	
96	60	Dump space size	Char(4)	
100	64	Dump data size	Char(4)	
104	68	Dump data size limit	Char(4)	
108	6C	Reserved	Char(20)	
128	80	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than eight causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total *number of bytes available* to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions are signaled in the event that the *receiver* contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception described previously.

The **dump space size** field is set with the current size value for the number of 512-byte blocks of space allocated for storage of dump data within the dump space.

The **dump data size** field is set with the current size value for the number of 512-byte blocks of dump data contained in the dump space. This value specifies the number of blocks from the start of the dump space through the block of dump data which has been placed into the dump space at the largest dump space offset value. A value of zero indicates that the dump space currently contains no dump data.

The **dump data size limit** field is set with the current size limit for the number of 512-byte blocks of dump data which may be stored in the dump space. A value of zero indicates that no explicit limitation is placed on the amount of dump data which may be stored in the dump space. The machine implicitly places a limit on the maximum size of a dump space. This value of this limitation is dependent upon the specific implementation of the machine.

Warning: Temporary Level 3 Header

Authorization Required

•

- Operational
  –
  – Operand 2
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Operand 2
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Exception Description (MATEXCPD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03D7	Attribute receiver	Exception description	Materialization option

Operand 1: Space pointer.

Operand 2: Exception description.

Operand 3: Character(1) scalar.

Description: The instruction materializes the attributes (operand 3) of an *exception description* (operand 2) into the *receiver* specified by operand 1.

The template identified by operand 1 must be a 16-byte aligned area in the space if the *materialization option* is hex 00.

Operand 2 identifies the *exception description* to be materialized.

The value of operand 3 specifies the *materialization option*. If the *materialization option* is hex 00, the format of the exception description materialization is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Template size	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Control flags	Char(2)	
8	8		Exception handling action	Bits 0-2

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			000 = Do not handle. (Ignore occurrence of exception and continue processing.)	
			001 = Do not handle. (Disable this exception description and continue to search this invocation for another exception description to handle the exception.)	
			010 = Do not handle. (Continue to search for an exception description by resignaling the exception to the preceding invocation.)	
			100 = Defer handling. (Save exception data for later exception handling.)	
			101 = Pass control to the specified exception handler.	
8	8		No data	Bit 3
			0 = Exception data is returned	
			1 = Exception data is not returned	
8	8		Reserved (binary 0)	Bit 4
8	8		User data indicator	Bit 5
			0 = User data not present	
			1 = User data present	
8	8		Reserved (binary 0)	Bits 6-7
8	8		Exception handler type	Bits 8-9
			00 = External entry point	
			01 = Internal entry point	
			10 = Branch point	
8	8		Reserved (binary 0)	Bits 10-15
10	A	Instruction number to be given control (if *exception handler type* is *internal entry point* or *branch point*; otherwise, 0)	UBin(2)	
12	C	Length of compare value (maximum of 32 bytes)	Bin(2)	
14	E	Compare value (size established by value of length of compare value field)	Char(32)	
46	2E	Number of exception IDs	Bin(2)	
48	30	System pointer to the exception handling program (if exception handler type is external entry point)	System pointer	
64	40	Pointer to user data (not present if value of user data indicator is 0)	Space pointer	
80	50	Exception ID (one for each exception ID dictated by the number of exception IDs field)	[*] Char(2)	
*	*	—- End —-		

If the *materialization option* is hex 01, the format of the materialization is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)

Offset Dec	Offset Hex	Field Name	Data Type and Length		
8	8	Control flags	Char(2)		
8	8		Exception handling action		Bits 0-2
			000 =	Do not handle. (Ignore occurrence of exception and continue processing.)	
			001 =	Do not handle. (Disable this exception description and continue to search this invocation for another exception description to handle the exception.)	
			010 =	Do not handle. (Continue to search for an exception description by resignaling the exception to the preceding invocation.)	
			100 =	Defer handling. (Save exception data for later exception handling.)	
			101 =	Pass control to the specified exception handler.	
8	8		No data		Bit 3
			0 =	Exception data is returned	
			1 =	Exception data is not returned	
8	8		Reserved (binary 0)		Bits 4-15
10	A	—- End —-			

If the *materialization option* is hex 02, the format of the materialization is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Compare value length (maximum of 32 bytes)	Bin(2)	
10	A	Compare value	Char(32)	
42	2A	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the receiver operand contains insufficient area for the materialization.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Independent Index Attributes (MATINXAT)

Op Code (Hex)	Operand 1	Operand 2
0462	Receiver	Index

Operand 1: Space pointer.

Operand 2: System pointer.

Bound program access

```
Built-in number for MATINXAT is 38.
MATINXAT (
          receiver   : address
          index      : address of system pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The instruction materializes the creation attributes and current operational statistics of the independent index identified by operand 2 into the space identified by operand 1. The format of the attributes materialized is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
8	8	Object identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Object creation options	Char(4)	
40	28		Existence attributes	Bit 0
			0 = Temporary	
			1 = Reserved	
40	28		Space attribute	Bit 1
			0 = Fixed-length	
			1 = Variable-length	
40	28		Context	Bit 2
			0 = Addressability not in context	
			1 = Addressability in context	
40	28		Access group	Bit 3
			0 = Not a member of access group	
			1 = Member of access group	
40	28		Reserved (binary 0)	Bits 4-12
40	28		Initialize space	Bit 13
			0 = Initialize	
			1 = Do not initialize	
40	28		Automatically extend space	Bit 14
			0 = No	
			1 = Yes	
40	28		Hardware storage protection level	Bits 15-16
			00 = Reference and modify allowed for user state programs	
			01 = Only reference allowed for user state programs	
			10 = Only reference allowed in any state	
			11 = No reference or modify allowed for user state programs	
40	28		Reserved (binary 0)	Bits 17-19
40	28		Always enforce hardware storage protection of this index	Bit 20

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = Enforce hardware storage protection of this index only when hardware storage protection is enforced for all storage.	
			1 = Enforce hardware storage protection of this index at all times.	
40	28		Always enforce hardware storage protection of the associated space	Bit 21
			0 = Enforce hardware storage protection of this space only when hardware storage protection is enforced for all storage.	
			1 = Enforce hardware storage protection of this space at all times.	
40	28		Reserved (binary 0)	Bits 22-31
44	2C	Reserved (binary 0)	Char(4)	
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
53	35		Space alignment	Bit 0
			0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space.	
			1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space.	
			Ignore the value of this field when the *machine chooses space alignment* field has a value of 1.	
53	35		Reserved (binary 0)	Bits 1-2
53	35		Machine chooses space alignment	Bit 3
			0 = The space alignment indicated by the *space alignment* field is in effect.	
			1 = The machine chose the space alignment most beneficial to performance, which may have reduced maximum space capacity. The alignment chosen is a multiple of 512. Ignore the value of the *space alignment* field.	
53	35		Reserved (binary 0)	Bit 4
53	35		Main storage pool selection	Bit 5

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = Process default main storage pool used for object.	
			1 = Machine default main storage pool used for object.	
53	35		Reserved (binary 0)	Bit 6
53	35		Block transfer on implicit access state modification	Bit 7
			0 = Transfer the minimum storage transfer size for this object.	
			1 = Transfer the machine default storage transfer size for this object.	
53	35		Reserved (binary 0)	Bits 8-31
57	39	Reserved (binary 0)	Char(7)	
64	40	Context	System pointer	
80	50	Access group	System pointer	
96	60	Index attributes	Char(1)	
96	60		Entry length attribute	Bit 0
			0 = Fixed-length entries	
			1 = Variable-length entries	
96	60		Immediate update	Bit 1
			0 = No immediate update	
			1 = Immediate update	
96	60		Key insertion	Bit 2
			0 = No insertion by key	
			1 = Insertion by key	
96	60		Entry format	Bit 3
			0 = Scalar data only	
			1 = Both pointers and scalar data	
96	60		Optimized processing mode	Bit 4
			0 = Optimize for random references	
			1 = Optimize for sequential references	
96	60		Maximum entry length	Bit 5
			0 = Maximum entry length is 120 bytes	
			1 = Maximum entry length is 2,000 bytes	
96	60		Index coherency tracking	Bit 6
			0 = Do not track index coherency	
			1 = Track index coherency	
96	60		Longer template	Bit 7

Offset Dec	Hex	Field Name	Data Type and Length
		0 = The template is the original size	
		1 = The template is longer	
97	61	Argument length	Bin(2)
99	63	Key length	Bin(2)
101	65	Index statistics	Char(12)
101	65	Entries inserted	UBin(4)
105	69	Entries removed	UBin(4)
109	6D	Find operations	UBin(4)
113	71	—- End —-	

If the bit **longer template** is set to binary 1, then the longer template is defined starting at offset 113 of the operand 1 template. The longer template is defined as follows:

Offset Dec	Hex	Field Name	Data Type and Length
113	71	Template version	Char(1)
114	72	Index format	Char(1)
		0 = Maximum object size of 4 Gigabytes.	
		1 = Maximum object size of 1 Terabyte.	
115	73	Reserved (binary 0)	Char(61)
176	B0	—- End —-	

The first 4 bytes of the materialization identify the total **number of bytes provided** that may be used by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged.

No exceptions other than the *materialization length invalid* (hex 3803) exception described previously are signaled in the event that the receiver contains insufficient area for the materialization.

The template identified by the operand 1 space pointer must be 16-byte aligned. Values in the template remain the same as the values specified at the creation of the independent index except that the *object identification*, *context*, *size of space*, *index attributes*, and *index statistics* contain current values.

If the *entry length* is *fixed*, then the *argument length* is the value supplied in the template when the index was created. If the *entry length* is *variable*, then the *argument length* field is equal to the length of the longest entry that has ever been inserted into the index.

The number of arguments in the index equals the number of **entries inserted** minus **entries removed**. The value of the **find operations** field is initialized to 0 each time the index is materialized. The value may not be correct after an abnormal system termination.

The field **template version** identifies the version of the longer template. It must be set to hex 00.

The **index format** field determines the format of the index. This attribute cannot be modified after the index has been created. If an index is created with a format of hex 01 (maximum size of 1 terabyte), the index cannot be saved to a target release earlier than Version 5 Release 2.

Warning: Temporary Level 3 Header

Authorization Required

•
- Operational
  –
  – Operand 2
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•
- Materialize
  –
  – Operand 2
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Instruction Attributes (MATINAT)

Op Code (Hex)	Operand 1	Operand 2
0526	Receiver	Selection template

Operand 1: Space pointer.

Operand 2: Character scalar.

Bound program access
Built-in number for MATINAT is 466. MATINAT (receiver : address selection_template : address)

Description: This instruction materializes the attributes of the non-bound program instruction that are selected in operand 2 and places them in the *receiver* indicated by operand 1.

Operand 2 is a 16-byte *selection template*. Only the first 16 bytes are used. Any excess bytes are ignored. Operand 2 has the following format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Selection template	Char(16)	
0	0		Invocation number	Bin(2)
2	2		Instruction number	Bin(4)
6	6		Reserved (binary 0)	Char(10)
16	10	—- End —-		

The **invocation number** is a specific identifier for the target invocation, in the thread, that is to be materialized. This program must be observable or the *program not observable* (hex 1E01) exception is signaled.

The **instruction number** specifies the instruction in the specified program invocation that is to be materialized.

Operand 1 addresses a 16-byte aligned template where the materialized data is placed. The format of the data is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided by the user	Bin(4)
4	4		Number of bytes available to be materialized	Bin(4)
8	8	Object identification	Char(32)	
8	8		Program type	Char(1)
9	9		Program subtype	Char(1)
10	A		Program name	Char(30)
40	28	Offset to instruction attributes	Bin(4)	
44	2C	Reserved (binary 0)	Char(8)	
52	34	Instruction attributes	Char(*)	
52	34		Instruction type	Char(2)
52	34		Instruction version	Bits 0-3
			Hex 0000 = 2-byte operand references	
			Hex 0001 = 3-byte operand references	
52	34		Reserved (binary 0)	Bits 4-1
54	36		Instruction length as input to Create Program	Bin(2)

Offset Dec	Offset Hex	Field Name	Data Type and Length		
56	38		Offset to instruction form specified as input to Create Program	Bin(4)	
60	3C		Reserved (binary 0)	Char(4)	
64	40		Number of instruction operands	Bin(2)	
66	42		Operand attributes offsets	Char(*)	
66	42		An offset is materialized for each of the operands of the instruction specifying the offset to the attributes for the operand		[*] Bin(4)
*	*		Instruction form specified as input to Create Program	Char(*)	
*	*		Instruction operation code		Char(2)
*	*		Optional extender field and operand fields		Char(*)
*	*		Operand attributes	Char(*)	
			A set of attributes following this format is materialized for each of the operands of the instruction. Compound operand references result in materialization of only one set of attributes for the operand which describe the substring or array element as is appropriate. See the specific format described below for each operand type.		
*	*		Operand type		Bin(2)
			1 = Data object		
			2 = Constant data object		
			3 = Instruction number reference		
			4 = Argument list		
			5 = Exception description		
			6 = Null operand		
			7 = Space pointer machine object		
*	*		Operand specific attributes		Char(*)
			See descriptions below for detailed formats. Nothing is provided for null operands.		
*	*	—- End —-			

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Data object	Char(32)		
		For a data object, the following operand attributes are materialized.			
0	0		Operand type = 1	Bin(2)	
2	2		Data object specific attributes	Char(7)	
2	2		Element type		Char(1)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			Hex 00 = Binary	
			Hex 01 = Floating-point	
			Hex 02 = Zoned decimal	
			Hex 03 = Packed decimal	
			Hex 04 = Character	
			Hex 08 = Pointer	
3	3		Element length	Char(2)
				If binary, or character, or floating-point:
3	3			Length
				If zoned decimal or packed decimal:
3	3			Fractional digits
3	3			Total digits
				If pointer:
3	3			Length = 16
5	5		Array size	Bin(4)
			If scalar, then value of 0.	
			If array, then number of elements.	
9	9		Reserved (binary 0)	Char(6)
15	F		Data object addressability	Char(17)
15	F		Addressability indicator	Char(1)
			Hex 00 = Addressability was not established	
			Hex 01 = Addressability was established	
16	10		Space pointer to the object if addressability could be established	Space pointer
32	20	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Constant data object	Char(*)	
		For a constant data object, the following operand attributes are materialized (immediate operands as constants, signed immediates as binary, and unsigned immediates as character).		
0	0		Operand type = 2	Bi
2	2		Constant specific attributes	Cl
2	2		Element type	
			Hex 00 = Binary	
			Hex 01 = Floating-point	
			Hex 02 = Zoned decimal	
			Hex 03 = Packed decimal	
			Hex 04 = Character	
3	3		Element length	
				If binary, or chara
3	3			Length
				If zoned decimal c
3	3			Fractional digits
3	3			Total digits
5	5		Reserved (binary 0)	
9	9		Reserved (binary 0)	Cl
16	10		Constant value	Cl
*	*	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Instruction references	Char(*)	
		For instruction references, either through instruction definition lists or immediate operands, the following operand attributes are materialized.		
0	0		Operand type = 3	Bin(2)
2	2		Number of instruction reference elements 1 = Single instruction reference >1 = Instruction definition list	Bin(2)
4	4		Reserved (binary 0)	Char(12)
16	10		Reference list The instruction number of each instruction reference is materialized in the order in which they are defined.	Char(*)
*	*	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Argument list	Char(*)
		For an argument list, the following operand attributes are materialized.	
0	0		Operand type = 4
2	2		Argument list specific attributes
2	2		Actual number of list entries
4	4		Maximum number of list entries
6	6		Reserved (binary 0)
16	10		Addressability to list entries
16	10		Space pointer to each list entry for the number of actual
			A value of all zeros is materialized if addressability coul
*	*	—- End —-	

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Exception description	Char(48)
		For an exception description, the following operand attributes are materialized.	
0	0		Operand type = 5
2	2		Reserved (binary 0)
12	C		Control flags
12	C		Exception handling action
			000 = Ignore occurrence of exception and continue p
			001 = Disabled exception description
			010 = Continue search for an exception description b
			100 = Defer handling
			101 = Pass control to the specified exception handler
12	C		Reserved (binary 0)
14	E		Compare value length
16	10		Compare value
48	30	—- End —-	

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Space pointer machine object	Char(32)
		For a space pointer machine object, the following operand attributes are materialized.	
0	0		Operand type = 7
2	2		Reserved (binary 0)
15	F		Pointer addressability
15	F		Pointer value indicator
			Hex 00= Addressability value is not valid
			Hex 01= Addressability value is valid
16	10		Space pointer data object containing the space pointer m
32	20	—- End —-	

The first 4 bytes of the materialization identify the total **number of bytes provided by the user** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available to be materialized**. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then excess bytes are unchanged.

The materialization available for an instruction depends on the execution status of the program that the instruction is in. If the program has not executed to the point of the instruction, little or no meaningful information about the instruction can be materialized. If the program executes the instruction multiple times, the materialization will vary with each execution.

No exceptions are signaled in the event that the *receiver* contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception described previously.

This instruction is valid only when the program instruction to be materialized is from a non-bound program. If the invocation indicated by operand 2 is for any other invocation type, then an *instruction not valid for invocation type* (hex 2C1C) exception is signaled.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1E Machine Observation

1E01 Program Not Observable

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C1C Instruction Not Valid for Invocation Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Invocation (MATINV)

Op Code (Hex)	Operand 1	Operand 2
0516	Receiver	Selection information

Operand 1: Space pointer.

Operand 2: Space pointer.

Bound program access

```
Built-in number for MATINV is 149.
MATINV (
        receiver                : address
        selection_information   : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The attributes of the invocation selected through operand 2 are materialized into the *receiver* designated by operand 1.

Operand 2 is a space pointer that addresses a template that has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Control information	Char(2)	
0	0		Template extension	Bit 0 +
			0 = Template extension is not present.	
			1 = Template extension is present.	
0	0		Invocation number	Bits 1-15
2	2	Offset to list of parameters	Bin(4) +	
6	6	Number of parameter ODV numbers	Char(2) +	
8	8	Offset to list of exception descriptions	Bin(4) +	
12	C	Number of exception description ODV numbers	Char(2) +	
14	E	Template extension (optional)	Char(14) +	
14	E		Offset to list of space pointer machine objects	Bin(4) +
18	12		Number of space pointer machine object ODV numbers	Char(2) +
20	14		Reserved (binary 0)	Char(8)
28	1C	—- End —-		

Note: Fields annotated with a (+) must be set to all binary 0s if the invocation is not for a non-bound program. Otherwise, a *template value invalid* (hex 3801) exception is signaled.

The **offset to list of space pointer machine objects**, **offset to list of parameters**, and the **offset to list of exception descriptions** are relative to the start of the operand 2 template. Each list is an array of Char(2)

ODV numbers. The **number of space pointer machine object ODV numbers**, **number of parameter ODV numbers**, and **the number of exception description ODV numbers** define the sizes of the arrays.

Operand 1 is a space pointer that addresses a 16-byte aligned template into which the materialized data is placed. The format of the data is:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided by the user	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Object identification	Char(32)	
8	8		Program type	Char(1)
9	9		Program subtype	Char(1)
10	A		Program name	Char(30)
40	28	Trace specification	Char(2)	
40	28		Invocation trace status	Bit 0
			0 = Not tracing new invocations	
			1 = Tracing new invocations	
40	28		Return trace	Bit 1
			0 = Not tracing returns	
			1 = Tracing returns	
40	28		Invocation trace propagation	Bit 2
			0 = Not propagating invocation trace	
			1 = Propagating invocation trace	
40	28		Return trace propagation	Bit 3
			0 = Not propagating return trace	
			1 = Propagating return trace	
40	28		Reserved (binary 0)	Bits 4-15
42	2A	—- End —-		

The following fields are returned only for non-bound program invocations.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
42	2A	Instruction number	UBin(2)	
44	2C	Offset to parameter values	Bin(4)	
48	30	Offset to exception description value	Bin(4)	
52	34	Offset to space pointer machine object values	Bin(4)	
		(Optional-This data is present only if the template extension is present in the selection information.)		
*	*	Space pointer machine objects	Char(*)	
		(Optional-This data is present only if the template extension is present in the selection information.)		
*	*		For each ODV number specified for a space pointer machine object, the value of the space pointer machine object is materialized as follows:	[*] Char(32)
*	*		Reserved (binary 0)	Cha
*	*		Pointer value indicator	Cha

Offset Dec	Offset Hex	Field Name	Data Type and Length		
			00 = Addressability value is not valid		
			01 = Addressability value is valid		
*	*		Space pointer data object containing the space pointer machine object value if addressability value is valid.		Space pointer
*	*	Parameters	Char(*)		
*	*		For each parameter ODT number specified, the address of the parameter data is materialized (If no parameter ODT numbers are materialized, this parameter is binary 0.)	[*] Space pointer	
*	*	Exception description	Char(*)		
*	*		For each exception description ODT number specified, the following is materialized:	[*] Char(36)	
*	*		Control flags		Char(2)
*	*			Exception handling action	
				000 =	Ignore occurren of exceptio and continue processi
				001 =	Disabled exceptio descripti
				010 =	Continue search for an exceptio descripti by resignali the exceptio to the immedia precedin invocatio
				100 =	Defer handling
				101 =	Pass control to the specified exceptio handler
*	*			Reserved (binary 0)	
*	*		Compare value length		Bin(2)
*	*		Compare value		Char(3
*	*	—- End —-			

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then excess bytes are unchanged.

No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the *receiver* contains insufficient area for the materialization.

The instruction number returned depends on how control was passed from the invocation:

Exit Type	Instruction Number
Call External	Locates the Call External instruction
Event	Locates the next instruction to execute
Exception	Locates the instruction that caused the exception

The space pointers that address parameter values are returned in the same order as the corresponding ODT numbers in the input array. The same is true for the exception description values.

If the *offset to list of parameters* or the *number of parameter ODT numbers* is 0, no parameters are returned and the *offset to parameters* value is 0. If any parameters are returned, they are 16-byte aligned. If the *offset to list of exception descriptions* or the *number of exception description ODT numbers* is 0, no exception descriptions are returned and the *offset to exception description values* are 0.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

1E Machine Observation

- 1E01 Program Not Observable

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

- 3201 Scalar Type Invalid
- 3202 Scalar Attributes Invalid

36 Space Management

- 3601 Space Extension/Truncation

38 Template Specification

- 3801 Template Value Invalid
- 3802 Template Size Invalid
- 3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Invocation Attributes (MATINVAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0533	Receiver	Invocation identification	Attribute selection template

Operand 1: Space pointer.

Operand 2: Character(48) scalar or null.

Operand 3: Space pointer.

Bound program access

```
Built-in number for MATINVAT is 125.
MATINVAT (
          receiver                       : address
          invocation_identification      : address OR
                                           null operand
          attribute_selection_template   : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note

It is recommended that you use attribute IDs 33, 34 and 35 for 8-byte invocation, activation and activation group marks, respectively, rather than attribute IDs 12, 13 and 14. 4-byte marks can wrap and produce unexpected results.

Description: The attributes specified by operand 3 of the invocation specified by operand 2 are materialized into the *receiver* specified by operand 1. In addition to specifying the attributes to be materialized, operand 3 controls how they are arranged in the operand 1 *receiver*.

Operand 1 is a space pointer to an area that is to receive the materialized attribute values. The format of this area is determined by the value of the *attribute selection template*.

Operand 2 identifies the source invocation whose attributes are to be materialized. It also identifies the originating invocation whose activation group access right to the source invocation's activation group is to be verified. If operand 2 is null, the invocation issuing the instruction is both the source invocation and the originating invocation.

Operand 3 is a space pointer to a template that selects the invocation attributes to be materialized and specifies how they are to be arranged in the *receiver* template.

Operand 2

The value specified by operand 2 identifies the source and originating invocations. This operand can be null (which indicates the current invocation is to be used for the source and originating invocations) or it can contain either a invocation pointer to an invocation or a null pointer (which indicates the current invocation).

Operand 2 has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Source invocation offset	Bin(4)
4	4	Originating invocation offset	Bin(4)
8	8	Invocation range (ignored)	Bin(4)
12	C	Reserved (binary 0)	Char(4)
16	10	Source invocation pointer	Invocation pointer
32	20	Reserved (binary 0)	Char(16)
48	30	—- End —-	

If a non-null pointer is specified for *source invocation pointer*, then operand 2 must be 16-byte aligned in the space.

Terminology:

Requesting invocation
The invocation executing the MATINVAT instruction. Note that, in many cases, this invocation belongs to a system or language run-time procedure/program, and the instruction is actually being executed on behalf of another procedure or program.

Originating invocation
The invocation on whose behalf the instruction is being executed. It may be necessary to identify this invocation since its ″activation group access rights″ may need to be checked. This allows, for example, the requesting invocation to be a system state invocation with the instruction still performing an ″activation group access rights″ check that reflects the rights of the user.

Source invocation
The invocation whose attributes are to be materialized.

Activation group access rights
The rights that invocations executing in one activation group may have to access and modify the resources of another activation group.

Field descriptions:

Source invocation offset
A signed numerical value indicating an invocation relative to the invocation located by the source invocation pointer. A value of zero denotes the invocation addressed by the source invocation pointer, with increasingly positive numbers denoting increasingly later invocations in the stack, and increasingly negative numbers denoting increasingly earlier invocations in the stack.

If the source invocation pointer is not valid or the invocation identified by this offset does not exist in the stack, an *invocation offset outside range of current stack* (hex 2C1A) exception will be signaled.

Originating invocation offset
A signed numerical value identifying the originating invocation relative to the current invocation. Since this is an offset relative to the current invocation, only zero or negative values are allowed.

If the invocation identified by this offset does not exist in the stack, an *invocation offset outside range of current stack* (hex 2C1A) exception will be signaled.

Invocation range
This field is used by FNDRINVN and is ignored by this instruction.

Source invocation pointer
An *invocation pointer* to an invocation. If null, then the current invocation is indicated.

If the pointer identifies an invocation in another thread, a *process object access invalid* (hex 2C11) exception will be signaled. If the invocation identified by this pointer does not exist in the stack, an *object destroyed* (hex 2202) exception will be signaled.

Activation group access rights checking: This instruction sometimes (depending on the attributes materialized) requires that activation group access rights to the activation group of the source invocation be verified. In such cases, the *originator offset* field of operand 2 identifies the invocation whose right of access is to be checked. (That is, it identifies the invocation which is considered to have originated the request and on whose behalf the instruction is being executed.)

If *originator offset* is not zero, then the activation group of the requesting invocation must have the right to access the activation group of the invocation identified by *originator offset*. This check is made whether or not access rights to the source invocation need to be checked.

In the event that appropriate access rights are not found, an *activation group access violation* (hex 2C12) exception is signaled.

Note: The originating invocation identified by the originating invocation offset must be equal to or "newer" than the invocation identified as the source invocation. Otherwise, an *invalid origin invocation* (hex 2C19) exception will be signaled.

Usage note: In cases where *source invocation pointer* is null, operand 2 may be a constant.

Operand 3

The *attribute selection template* has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Selection template header	Char(16)
0	0		Number of attributes
4	4		Control flags
4	4		Attribute index indirect **0 =** *Offset to attribute index* specifies directly the lo **1 =** *Offset to attribute index* specifies the location of value
4	4		Reserved (binary 0)
5	5		Reserved (binary 0)
8	8		Offset to attribute index
12	C		Length of attribute index
16	10	Attribute selection entries	Char(*)
*	*	—- End —-	

The **attribute selection entries** are each 16 bytes long and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Attribute ID	Bin(4)
4	4	Control flags	Char(1)
4	4		Indirect Bit 0 **0 =** *Offset to receiver* specifies directly the location of the attribute value **1 =** *Offset to receiver* specifies the location of a space pointer which in turn specifies the location of the attribute value

Offset Dec	Hex	Field Name	Data Type and Length		
4	4		Return length		Bit 1
			0 =	A length field is not present with the attribute	
			1 =	A length field precedes the attribute	
4	4		Return status		Bit 2
			0 =	A status field is not present with the attribute	
			1 =	A status field precedes the attribute	
4	4		Pad		Bit 3
			0 =	No pad field is assumed to precede the attribute	
			1 =	A pad field of zero, eight, or twelve bytes is assumed to precede the attribute	
4	4		Reserved (binary 0)		Bits 4-7
5	5	Reserved (binary 0)	Char(3)		
8	8	Offset to receiver	Bin(4)		
12	C	Length of receiver	Bin(4)		
16	10	—- End —-			

Basic structure: The *attribute selection template* allows the user of MATINVAT considerable flexibility in deciding what invocation attributes are to be materialized and where their materializations are to be returned. This flexibility is achieved by having the *attribute selection template* consist of a header, followed by a series of entries, each of which identifies an attribute to be materialized, the location where it is to be materialized, and the amount of space reserved for its materialization.

The template header specifies the number of attribute entries present in the template, and it also allows the specification of an optional *attribute index* field. The *attribute index* field, if present, identifies the first *attribute selection entry* to be processed (causing entries prior to that one to be skipped). In addition, if the *attribute index* field is present, it is updated upon the normal or abnormal completion of the instruction to contain either zero (if completion is normal) or the number of the entries being processed (if the instruction ends with an exception).

Each *attribute selection entry* identifies the attribute to be materialized and the area where the materialization is to be returned. The attribute may be returned directly into the area addressed by the operand 1 space pointer, or it may be returned into an area addressed by a space pointer which is, in turn, contained in the area addressed by the operand 1 space pointer. These two cases are distinguished by the *indirect* bit.

In addition, each *attribute selection entry* contains:

-
- An offset value which is the offset relative to the operand 1 space pointer where either the attribute's materialization area or the pointer to the attributes' materialization area is contained.
- A length value identifying the maximum number of bytes of data to be materialized for the attribute.
- A flag indicating whether the length of the attribute is to be materialized.
- A flag indicating whether the status of the attribute is to be materialized.

- A flag indicating whether a pad field precedes the attribute (or its pointer, if *indirect* is specified). If present, the length of this "pad" field is automatically adjusted so that the combined length of the length, status, and pad fields is either zero or 16, maintaining the relative quadword alignment of the modification value if the length and/or status fields are present.

Note that, for the sake of regularity, the fields of the *attribute selection template* header are arranged in the same general fashion as those in the *attribute selection entries*.

Field descriptions:

Number of attributes
Specifies how many 16-byte *attribute selection entries* follow.

Attribute index indirect
If *attribute index indirect* is binary 0, then *offset to attribute index* specifies the location where the attribute index is stored as an offset from the location addressed by the operand 1 space pointer. If *attribute index indirect* is binary 1, then the location identified by *attribute index offset* must be quadword aligned and must contain a space pointer. This space pointer in turn addresses the location where the attribute index value is stored.

Offset to attribute index
Specifies the offset to the attribute index or the offset to a pointer to the attribute index, depending on the value of *attribute index indirect*.

Length of attribute index
Specifies the length of the area where the attribute index value is stored. This field must have a value of either zero or four.

If this field has a value of zero, then the first attribute entry to be processed is the first attribute entry in the template, and no feedback is given as to which attribute entry was being processed at the time of an exception. *Attribute index indirect* and *attribute index offset* are ignored.

If this field has a value of four, then the value of the attribute index, treated as a signed bin(4) value, must be greater than or equal to one and less than or equal to *number of attributes*. In this case the attribute index identifies the attribute entry to be processed first (with the first entry in the template having an index of one), and, in the event of an exception, the attribute index value is modified by this instruction such that it identifies the attribute entry being processed at the time of the exception. If the instruction completes without an exception, then the attribute index value is set to zero.

Attribute ID
Specifies the attribute to be materialized. Values that may be specified are:

1	Invocation pointer to specified invocation. (16 bytes, quadword aligned.)
2	Automatic storage pointer. Space pointer to the automatic storage for this invocation. If no automatic storage exists for this invocation, then a null pointer is returned. (16 bytes, quadword aligned, access rights required.)
3	Static storage pointer. Space pointer to the static storage for a non-bound program invocation, if any exists. Otherwise, a null pointer value is returned. (16 bytes, quadword aligned, access rights required.) **Note:** For bound program procedure invocations there is no single "distinguished" static storage area, but instead there may be multiple static storage areas. The list of static storage areas corresponding to the invocation's activation can be obtained by using the Materialize Activation Attributes (MATACTAT) instruction.

4 Parameter list pointer. Space pointer to the parameter list passed to this invocation (bound program procedure invocations only). If the procedure for this invocation does not have a parameter list, or if this invocation is for a program entry procedure or a non-bound program, then a null pointer value is returned. (16 bytes, quadword aligned, access rights required.)

6 Program pointer. System pointer to the program for this invocation. If the program no longer exists then a null pointer is returned. (16 bytes, quadword aligned, access rights required.)

7 Space pointer to *module* associated space. For bound program procedures, this space pointer addresses the secondary associated space in the bound program that was propagated from the primary associated space of the bound program module. For non-bound programs, this space pointer addresses the program's primary associated space. If the appropriate associated space does not exist in the program or if the program no longer exists, then a null pointer is returned. For both bound and non-bound program invocations the requesting invocation must have space authority to the program. (16 bytes, quadword aligned, access rights required.)

8 Pointer to containing scope. If the specified invocation is in a nested scope, then this is an invocation pointer to the invocation of the containing scope. Otherwise a null pointer is returned. (16 bytes, quadword aligned.)

9 Relative invocation offset to containing scope. If the specified invocation is in a nested scope, then this is the relative invocation offset to the invocation of the containing scope. Otherwise, a value of zero is returned. Note that the relative invocation offset will be a negative number and is relative to the specified invocation. (4 bytes.)

10 Lexical level number. Outer procedures have a lexical level number of 1. (4 bytes.)

11 Invocation number. (2 bytes.)

12 Invocation mark. (4 bytes.)

13 Activation mark. If no activation exists for this invocation, then a zero value is returned. (4 bytes.)

14 Activation group mark. (4 bytes.)

If the activation resides in a shared activation group owned by another process, or if no activation exists for the invocation, then the value returned is as follows:

1 if this is a system state invocation

2 if this is a user state invocation

15
Invocation type. The possible values for invocation type are:

Hex 01 =
Call external

Hex 02 =
Transfer control

Hex 03 =
Event handler

Hex 04 =
External exception handler (for non-bound program)

Hex 05 =
Initial program in process problem state

Hex 06 =
Initial program in process initiation state

Hex 07 =
Initial program in process termination state

Hex 08 =
Invocation exit (for non-bound program)

Hex 09 =
Return or return/XCTL trap handler

Hex 0A =
Call program

Hex 0B =
Cancel handler (bound program only)

Hex 0C =
Exception handler (bound program only)

Hex 0D =
Call bound procedure/call with procedure pointer

Hex 0E =
Process Default Exception Handler

(1 byte.)
16
Routine type. The possible values for routine type are:

Hex 01 =
Non-Bound Program

Hex 02 =
Bound Program Entry Procedure (PEP)

Hex 03 =
Bound Program Procedure

Note: Bound program procedures are contained within bound programs, bound service programs, and Java programs. All discussion of bound program procedure semantics also apply to Java program procedures.

(1 byte.)
17
State invocation was invoked with. (2 bytes.)

Hex 8000 =
System state

Hex 0001 =
User state

18
State for invocation. (2 bytes.)

Hex 8000 =
System state

Hex 0001 =
User state

19
Invocation status of the specified invocation (including invocation flags).

Bit 0

Cancelled

Bit 1

Ending — a return operation has been initiated from within the invocation or the actual termination of a cancelled invocation has begun.

Bit 2

Invocation interrupted by exception

Bit 3

Invocation interrupted by event (reserved)

Bit 4

Invocation is a non-bound program CALLX exception handler

Bit 5

Invocation contains a non-bound program CALLI exception handler

Bit 6

Invocation contains a signalled non-bound program branchpoint handler

Bit 7

Retry not allowed

Bit 8

Resume not allowed

Bit 9

Resume point has been modified

Bit 10

Invocation is a program entry procedure and is marked as the oldest in the activation group (This is also known as a hard control boundary.)

Bit 11

Invocation is a soft control boundary.

Bit 12

Invocation created an unnamed activation group.

Bits 13-15

Reserved

Bits 16-31

Invocation flags

(4 bytes.)

Performance consideration: When the only invocation status information required is the invocation flags, there may be a significant performance advantage if the following attribute is materialized instead of this one.
20
Invocation flags of the specified invocation. This attribute has the same format as the *invocation status* attribute, except that the first two bytes are returned as zero. (4 bytes.)
23
Cancel reason of the specified invocation. (4 bytes.)
24
Suspend point. Suspend pointer identifying the location within the invocation's routine where execution was suspended due to a call, interrupt, or machine operation. If the program no longer exists then a null pointer is returned. (16 bytes, quadword aligned, access rights required.)
25
Resume point.

A suspend pointer identifying the location within the invocation's routine where execution will resume if execution is allowed to resume in the invocation. If the invocation is suspended for some cause that permits resumption, then this is initially set to the location that logically follows the suspend point. If the invocation is suspended for some cause that does not permit resumption, then this is initially set to be a null pointer. If the resume point is modified via Modify Invocation Attributes then a suspend pointer (or null pointer) corresponding to the modified resume point is returned. If the program no longer exists or if the invocation is cancelled or ending, then a null pointer is returned. (16 bytes, access rights required, quadword aligned.)
26
Interrupt message invocation. If the invocation is interrupted due to an exception interrupt, and the message causing the interrupt has not been removed or modified to a non-interrupt state, then this is an invocation pointer which addresses the invocation to which the interrupt message is enqueued. If no interrupt cause currently exists, then a null pointer is returned. (16 bytes, quadword aligned.)
27
Interrupt message reference key. If the invocation is interrupted due to an exception interrupt, and the message causing the interrupt has not been removed or modified to a non-interrupt state, then this is the message reference key of the interrupt cause message. If no interrupt cause currently exists, then a value of zero is returned. (4 bytes.)
28
External exception handler's monitoring invocation. If the specified invocation is an external exception handler for a non-bound program, then this is an invocation pointer identifying the invocation which enabled the handler (also the invocation where the exception message is currently enqueued). Otherwise, a null pointer is returned. (16 bytes, quadword aligned.)
29
External exception handler's message reference key. If the specified invocation is an external exception handler for a non-bound program, then this is the message reference key of the corresponding exception message. Otherwise, a zero value is returned. (4 bytes.)
30
Non-bound program internal exception handler's message reference key. If the specified invocation is a non-bound program invocation with an internal exception handler active, then this is the message reference key of the exception message corresponding to the currently active internal exception handler. Otherwise, a zero value is returned. (4 bytes.)
31
Non-bound program branchpoint exception handler's message reference key. If the specified invocation is a non-bound program invocation with a branchpoint exception handler in a signalled state, then this is the message reference key of the exception message corresponding to the most recently signalled branchpoint exception exception handler. Otherwise, a zero value is returned. (4 bytes.)
32
Trap handler's message reference key. If the specified invocation was invoked as a trap handler, then this is the message reference key of the corresponding trap message. (Note that the trapped invocation is, by definition, the immediately preceding invocation.) Otherwise, a zero value is returned. (4 bytes.)
33
Invocation mark. (8 bytes.)
34
Activation mark. If no activation exists for this invocation, then a zero value is returned. (8 bytes.)
35
Activation group mark. (8 bytes.)

If the activation resides in a shared activation group owned by another process, or if no activation exists for the invocation, then the value returned is as follows:

1 if this is a system state invocation

2 if this is a user state invocation

Where "access rights required" is specified above, the activation group of the invocation identified as the originating invocation must have activation group access rights to the activation group of the source invocation or else an *activation group access violation* (hex 2C12) exception is signaled.

The invocation with an invocation number of 1 is always the first invocation in the stack.

Indirect If *indirect* is binary 0, then *offset to receiver* specifies the location where the selected attribute value is to be materialized as as offset from the location addressed by operand 1. If *indirect* is binary 1, then the location identified by *offset to receiver*, after accounting for any length, status, or pad fields specified, must be quadword aligned and must contain a space pointer. This space pointer in turn addresses the location where the selected attribute value is to be materialized.

Return length
If *return length* and *return status* are both binary 0, then only the attribute itself is materialized. If *return length* is binary 1, then the attribute (or attribute pointer, if *indirect* is true) is preceded by a four-byte value which specifies the length of the attribute (exclusive of the length value itself, and the status and pad fields, if present).

Return status
If *return status* is binary 1, then the attribute (or attribute pointer, if *indirect* is true) is preceded by a four-byte value which contains the status of the attribute.

If the status value is returned, it has the following format:

Bits 0-2	Reserved (binary 0)
Bit 3	Attribute unavailable at this time. (Eg, asking for the system pointer to a destroyed non-bound program.) The result returned is zeros for the minimum length defined.
Bit 4	Attribute not defined in this context. (Eg, asking for lexical level number from non-bound program invocation.) The result returned is zeros for the minimum length defined.
Bit 5	Attribute not defined at this time. (Eg, asking for interrupt message invocation when the invocation is not interrupted.) The result returned is zeros for the minimum length defined.
Bit 6	Attribute defined but null. (Eg, when asking for the resume point for an invocation for which resume is not currently allowed.) The result returned is zeros for the minimum length defined.
Bit 7	Attribute truncated. Indicates that the specified *length of receiver* was too small to allow the entire attribute to be returned. The truncated result is returned, as described earlier.
Bits 8-31	Reserved (binary 0)

If *return length* and *return status* are both binary 1 then the length field comes first, followed immediately by the status field.

Pad If either *return length* or *return status* is binary 1, and *pad* is also binary 1, then twelve bytes of pad are assumed between the length or status value and the attribute (or attribute pointer, if *indirect* is true). If both *return length* and *return status* are binary 1, and *pad* is also binary 1, then eight bytes of pad are assumed between the status value and the attribute (or attribute pointer). If *return length* and *return status* are both binary 0, then no padding occurs, regardless of the value of *pad*. The area occupied by the pad is not modified by this instruction.

Note: *Pad* makes it easier to quadword align the area to receive the materialized attribute (if *indirect* is false) or the area containing the attribute pointer (if *indirect* is true) when *return status* and/or *return length* are also specified.

Offset to receiver
Specifies the offset to the location where the selected attribute value is to be materialized, or the offset to a pointer to the location, depending on the value of *indirect*.

Length of receiver
Specifies the length of the area where the attribute value is to be materialized.

This length indicates the length of the actual area available for materializing the attribute, and *does not* include the length of any length, status, or pad field. If the number of bytes of attribute data available to be materialized (exclusive of the status, length, and pad fields, if any) exceeds *length of receiver*, then only *length of receiver* bytes of data are returned. No exception is signalled in this case.

If *indirect* is a binary 0, then *length of receiver* indicates the length of the area located by *offset to receiver*. If *indirect* is a binary 1, then *length of receiver* indicates the length of the area located by the indirect space pointer identified by *offset to receiver*.

In the case that *length of receiver* is sufficient to receive only part of a field in an attribute structure, then the partial field may or may not be materialized.

Individual attribute entries are processed in order, with the attributes specified by each entry being materialized before processing of the next entry begins. If an exception occurs while processing an attribute entry, then the attributes materialized due to the preceding attribute entries will still be present in their specified result locations.

For attributes which include pointers, the specified direct or indirect value location, after accounting for any length, status, or pad fields, must be quadword aligned or a *boundary alignment* (hex 0602) exception may occur. (The exception is not guaranteed to occur, eg, in the case where *length of receiver* is insufficient to include the materialized pointer, or when a null pointer is returned.)

If the value locations of individual attribute entries overlap, then the values will be overlaid in the sequence implied by the attribute entry order. If the value location of a non-indirect result overlays the location of the space pointer for an indirect result, then the validity of the space pointer will depend on the order of the associated entries.

Warning: Temporary Level 3 Header

Authorization Required

•

- Activation group access
  –
  – From the activation group of the invocation issuing the instruction to the activation group of the originating invocation identified by operand 2
  – When an attribute annotated with ″access rights required″ is specified: From the activation group of the originating invocation identified by operand 2 to the activation group of the source invocation identified by operand 2
- Space authority
  –
  – For the module associated space option, the requesting invocation must have space authority to the program executing in the source invocation identified by operand 2

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2C Program Execution

- 2C11 Process Object Access Invalid
- 2C12 Activation Group Access Violation
- 2C19 Invalid Origin Invocation
- 2C1A Invocation Offset Outside Range of Current Stack

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

- 3203 Scalar Value Invalid

36 Space Management

- 3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Invocation Entry (MATINVE)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0547	Receiver	Selection information	Materialization options

Operand 1: Character variable scalar.

Operand 2: Character(8) scalar or null.

Operand 3: Character(1) scalar or null.

Bound program access

```
Built-in number for MATINVE is 479.
MATINVE (
         receiver                 : address
         materialization_options  : unsigned binary(4) literal
)
```

Note: There is no operand to identify an invocation. Only the current invocation can be materialized with this instruction. Thus the operand 2 description below does not apply.

Note: The *materialization options* operand must be specified. It is referred to as operand 3 in the description below.

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note

It is recommended that you use Short Materialization Type 6 for an 8-byte invocation mark rather than Short Materialization Type 2 and that you use the 8-byte invocation mark and the 8-byte thread mark counter from the end of the Long Materialization receiver. 4-byte marks can wrap and produce unexpected results.

Description: This instruction materializes the attributes of the specified invocation entry within the thread issuing the instruction. The attributes specified by operand 3 of the invocation selected through operand 2 are materialized into the *receiver* designated by operand 1.

Operand 2 is an 8-byte template or a null operand. If operand 2 is null, it indicates that the attributes of the current invocation are to be materialized. If operand 2 is not null, it must be an 8-byte template which specifies the invocation to be materialized. Only the first 8 bytes are used. Any excess bytes are ignored. It has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Selection information	Char(8)	
0	0		Relative invocation number	Char(2)
2	2		Reserved	Char(6)

Offset Dec	Hex	Field Name	Data Type and Length
8	8	—- End —-	

If operand 2 is not null, it is restricted to a constant with the **relative invocation number** field specifying a value of zero, which indicates that the attributes of the current invocation are to be materialized.

Operand 3 is a 1-byte value or a null operand. If operand 3 is null, it indicates that the attributes for a *materialization options* value of hex 00 are to be materialized. If operand 3 is not null, it must be a 1-byte value which specifies the type of materialization to be performed. Option values that are not defined below are reserved values and may not be specified. Only the first byte is used. Any excess bytes are ignored. It has the following format:

Offset Dec	Hex	Field Name	Data Type and Length
0	0	Materialization options	Char(1)
		Hex 00 = Long materialization	
		Hex 01 = Short materialization type 1	
		Hex 02 = Short materialization type 2	
		Hex 03 = Short materialization type 3	
		Hex 04 = Short materialization type 4	
		Hex 05 = Short materialization type 5	
		Hex 06 = Short materialization type 6	
1	1	—- End —-	

If operand 3 is not null, it is restricted to a constant character scalar or an immediate value.

Operand 1 specifies a *receiver* into which the materialized data is placed. It must specify a character scalar with a minimum length which is dependent upon the *materialization options* specified for operand 3. If the length specified for operand 1 is less than the required minimum, an exception is signaled. Only the bytes up to the required minimum length are used. Any excess bytes are ignored. For the *materialization options* which produce pointers in the materialized data, 16-byte space alignment is required for the *receiver*. The data placed into the *receiver* differs depending upon the *materialization options* specified. The following descriptions detail the formats of the optional materializations.

Long Materialization: For a *materialization options* value of hex 00, the minimum length for the *receiver* is 144 bytes. It has the following format:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Long materialization	Char(144)	
0	0		Reserved	Char(12)
12	C		Thread mark counter	Bin(4)
16	10		Reserved	Char(32)
48	30		Associated program pointer (zero for data base select/omit program)	System pointer
64	40		Invocation number	Bin(2)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
66	42		Invocation type	Char(1)
			Hex 00= Data base select/omit program	
			Hex 01 = Call external	
			Hex 02 = Transfer control	
			Hex 03 = Event handler	
			Hex 04 = External exception handler	
			Hex 05 = Initial program in process problem state	
			Hex 06 = Initial program in process initiation state	
			Hex 07 = Initial program in process termination state	
			Hex 08 = Invocation exit	
			Hex 09 = Return trap handler or return/XCTL trap handler	
			Hex 0A = Call program	
			Hex 0B = Reserved	
			Hex 0C = Reserved	
			Hex 0D = Reserved	
			Hex 0E = Process Default Exception Handler	
67	43		Reserved (binary 0)	Char(1)
68	44		Invocation mark	Bin(4)
72	48		State invocation was invoked with	Char(2)
			Hex 8000 = System state	
			Hex 0001 = User state	
74	4A		State for invocation	Char(2)
			Hex 8000 = System state	
			Hex 0001 = User state	
76	4C		Reserved	Char(4)
80	50		Automatic storage frame (ASF) pointer	Space pointer
96	60		Static storage frame (SSF) pointer	Space pointer
112	70		Invocation mark	UBin(8)

Offset Dec	Hex	Field Name	Data Type and Length		
			For Non-Bound programs, the following datatype should be used:		
112	70		Invocation mark (Non-Bound program)		Cha
120	78		Thread mark counter	UBin(8)	
			For Non-Bound programs, the following datatype should be used:		
120	78		Thread mark counter (Non-Bound program)		Cha
128	80		Reserved	Char(16)	
144	90	—- End —-			

Short Materialization Type 1: For a *materialization options* value of hex 01, the minimum length for the receiver is 16 bytes. It has the following format:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Short materialization type 1	Char(16)	
0	0		Associated program pointer (null for data base select/omit program)	System pointer
16	10	—- End —-		

Short Materialization Type 2: For a *materialization options* value of hex 02, the minimum length for the *receiver* is 4 bytes. It has the following format:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Short materialization type 2	Char(4)	
0	0		Invocation mark	Bin(4)
4	4	—- End —-		

Short Materialization Type 3: For a *materialization options* value of hex 03, the minimum length for the *receiver* is 16 bytes. It has the following format:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Short materialization type 3	Char(16)	
0	0		ASF (Automatic Storage Frame) pointer	Space pointer
16	10	—- End —-		

Short Materialization Type 4: For a *materialization options* value of hex 04, the minimum length for the *receiver* is 16 bytes. It has the following format:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Short materialization type 4	Char(16)	
0	0		SSF (Static Storage Frame) pointer	Space pointer
16	10	—- End —-		

Short Materialization Type 5: For a *materialization options* value of hex 05, the minimum length for the *receiver* is 4 bytes. It has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Short materialization type 5	Char(4)	
0	0		State invocation was invoked with	Char(2)
2	2		State for invocation	Char(2)
4	4	—- End —-		

Short Materialization Type 6: For a *materialization options* value of hex 06, the minimum length for the *receiver* is 8 bytes. It has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Short materialization type 6	Char(8)	
0	0		Invocation mark	UBin(8)
			For Non-Bound programs, the following datatype should be used:	
0	0		Invocation mark (Non-Bound program)	Char(8)
8	8	—- End —-		

The **thread mark counter** is a thread-specific counter maintained by the machine. It is used to assign **invocation marks** within the thread. The current value of the *thread mark counter* at the time the instruction executes is returned in the 8-byte *thread mark counter* field. The low order 4 bytes is returned in the 4-byte *thread mark counter* field.

The **associated program pointer** is a system pointer that locates the program associated with the invocation entry.

The **invocation number** is the stack depth of the invocation within the invocation stack. The *invocation number* of a new invocation entry is one more than that in the calling invocation. The first invocation in the current thread has an invocation number of one.

The **invocation type** indicates how the associated program was invoked.

The **invocation mark** identifies the invocation within the thread.

The **state invocation was invoked with** value represents the state in which the machine was running when the program was called or transferred to.

The **state for invocation** value represents the state in which the machine is running the program.

The **ASF (Automatic Storage Frame) pointer** is a space pointer that is set to address the start of the ASF associated with the invocation. The associated program's automatic data starts 64 bytes after the area addressed by this pointer.

The **SSF (Static Storage Frame) pointer** is a space pointer that is set to address the start of the static storage frame (SSF) associated with the invocation. The associated program's static data starts 64 bytes after the area addressed by this pointer. This pointer will be set to a value of all zeros if the invoked program does not have static data.

The fields labeled reserved in the descriptions of the optional materializations are currently reserved for future use. These fields may be altered by this instruction depending upon the particular implementation of the machine. Any values set into these fields are meaningless.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Invocation Stack (MATINVS)

Op Code (Hex)	Operand 1	Operand 2
0546	Receiver	Process

Operand 1: Space pointer.

Operand 2: System pointer or null.

Bound program access

```
Built-in number for MATINVS is 150.
MATINVS (
         receiver   : address
         process    : address of system pointer OR
                      null operand
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note

It is recommended that you use Materialize Invocation Attributes (MATINVAT) or Materialize Invocation Entry (MATINVE) to materialize an 8-byte invocation mark. 4-byte marks can wrap and produce unexpected results.

Description: The invocation stack of either the current thread, or the initial thread of another process is materialized. The materialization starts with the oldest invocation and proceeds toward the newest invocation.

Invocation stack entry attributes from the selected thread are returned in the template specified by operand 1. Operand 2 identifies the selected thread as follows. If operand 2 is either null or a system pointer to the process control space of the current process, then the invocation stack of the current thread will be materialized. If operand 2 identifies a process other than the current process, then the invocation stack of the initial thread of that process will be materialized. In this latter case the process must be the original initiator of the target process, or must have process control special authorization.

Operand 1 is a space pointer that addresses a 16-byte aligned template into which is placed the materialized data. The format of the data is:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided by the user	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Number of invocation entries	Bin(4)	
12	C	Thread mark counter	Bin(4) +	
16	10	Invocation entries	[*] Char(128)	
		(An invocation entry is materialized for each of the invocations currently on the invocation stack of the specified process.)		
*	*	—- End —-		

The invocation entries materialized are each 128 bytes long and have the following format:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Reserved	Char(32)
32	20	Associated program pointer	System pointer
		(null pointer value for a destroyed program)	
48	30	Invocation number	Bin(2)
50	32	Invocation mechanism	Char(1)

Offset Dec	Offset Hex	Field Name	Data Type and Length
		Hex 01 = Call external	
		Hex 02 = Transfer control	
		Hex 03 = Event handler	
		Hex 04 = External exception handler (for non-bound program)	
		Hex 05 = Initial program in process problem state	
		Hex 06 = Initial program in process initiation state	
		Hex 07 = Initial program in process termination state	
		Hex 08 = Invocation exit (for non-bound program)	
		Hex 09 = Return or return/XCTL trap handler	
		Hex 0A = Call program	
		Hex 0B = Cancel handler (bound program only)	
		Hex 0C = Exception handler (bound program only)	
		Hex 0D = Call bound procedure/call with procedure pointer	
		Hex 0E = Process Default Exception Handler	
51	33	Invocation type	Char(1)
		Hex 01 = Non-Bound Program	
		Hex 02 = Bound Program Entry Procedure (PEP)	
		Hex 03 = Bound Program Procedure	
		Note: Bound program procedures are contained within bound programs, bound service programs, and Java programs. All discussion of bound program procedure semantics also apply to Java program procedures.	
52	34	Invocation mark	Bin(4) +
56	38	Instruction identifier (zero for destroyed, damaged, or suspended program)	Bin(4)
60	3C	Activation group mark	Bin(4) +

Offset Dec	Hex	Field Name	Data Type and Length
		This is the mark of the activation group which owns the activation associated with the invocation. However, if no activation exists for the invocation, or if an activation exists and it resides in a shared activation group owned by another process, then the activation group mark is returned as follows: 1 for a system state invocation 2 for a user state invocation	
			64 40 Suspend point Suspend pointer (null pointer value for destroyed program) 80 50 Reserved Char(48) 128 80 —- End —-

Note: Fields annotated with a plus sign (+) are not materialized if operand 2 identifies a process other than the current process. Fields not materialized are set to binary 0s.

The **number of invocation entries** value specifies the number of invocation entries available to be materialized. The **thread mark counter** is a thread-specific counter maintained by the machine. It is used to assign **invocation marks** within the thread. The value is the low order 4 bytes of the current value of the *thread mark counter* at the time the instruction executes.

The **associated program pointer** is a system pointer that locates the program associated with the invocation entry.

The **invocation number** is a number that uniquely identifies each invocation in the invocation stack. When an invocation is allocated, the invocation number of the new invocation entry is one more than that in the calling invocation. The first invocation in the current process state has an invocation number of one.

The **invocation type** indicates how the associated program was invoked.

The **invocation mark** identifies the invocation within the thread.

If the invocation type is a non-bound program the *instruction identifier* field will contain the instruction number which specifies the number of the instruction last being executed when the invocation passed control to the next invocation on the stack. If the invocation type is a bound program entry or a procedure, the *instruction identifier* field will contain the statement identifier, which is a compiler supplied number which allows the compiler to identify the source statement associated with a particular sequence of instructions.

Note: If the program is damaged or destroyed or if a statement identifier was not supplied by the compiler, a value of 0 is set.

The **suspend point** is a suspend pointer which identifies the instruction last being executed when the invocation passed control to the next invocation on the stack.

The fields labeled reserved are currently reserved for future use. These fields may be altered by this instruction depending upon the particular implementation of the machine. Any values set into these fields are meaningless.

The first 4 bytes of the materialization identifies the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identifies the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, the excess bytes are unchanged.

No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the materialization length exception described previously.

When the materialization is performed for a thread in a different process, the instruction attempts to interrogate and snapshot the invocation stack of the other thread concurrently with the ongoing execution of that thread. In this case, the interrogating thread and subject thread may have interleaving usage of the processor resource. Due to this, the accuracy and integrity of the materialization is relative to the state, static or dynamic, of the invocation stack in the subject thread over the time of the interrogation. If the invocation stack in the subject thread is in a very static state, not changing over the period of interrogation, the materialization may represent a good approximation of a snapshot of its invocation stack. To the contrary, if the invocation stack in the subject thread is in a very dynamic state, radically changing over the period of interrogation, the materialization is potentially totally inaccurate and may describe a sequence of invocations that was never an actual sequence that occurred within the thread. In addition to the above exposures to inaccuracy in attempting to take the snapshot, the ongoing status of the invocation stack of the subject thread may substantially differ from that reflected in the materialization, due to its continuing execution after completion of this instruction.

When the materialization is performed for the current thread, it does provide an accurate reflection of its invocation stack. In this case, concurrent execution of this instruction with execution of other instructions in the thread is precluded.

Warning: Temporary Level 3 Header

Authorization Required

•

- Process control special authorization
  –
  – For materializing a thread in a different process.
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialization
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

28 Process/Thread State

2802 Process Control Space Not Associated with a Process

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Journal Port Attributes (MATJPAT)

Op Code (Hex)	Operand 1	Operand 2
05A6	Receiver	Journal port or materialize template

Operand 1: Space pointer.

Operand 2: System pointer or space pointer data object.

Bound program access

```
Built-in number for MATJPAT is 84.
MATJPAT (
         receiver                                 : address
         journal_port_or_materialize_template     : address of system pointer OR
                                                    address of space pointer(16)
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction materializes the creation attributes of the *journal port* specified by operand 2 and places the attributes in the *receiver* specified by operand 1.

The format of the materialization data is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization length	Char(8)	
0	0		Number of bytes provided by user	Bin(4)
4	4		Number of bytes available to be materialized	Bin(4)
8	8	Object identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Object creation options	Char(4)	
40	28		Existence attributes (binary 1)	Bit 0
40	28		Space attributes **0 =** Fixed-length **1 =** Variable-length	Bit 1
40	28		Initial context	Bit 2
40	28		Access group	Bit 3
40	28		Replace option	Bit 4
40	28		Reserved	Bits 5-12

Offset Dec	Offset Hex	Field Name	Data Type and Length	
40	28		Initialize space	Bit 13
40	28		Reserved	Bits 14-18
40	28		Use system storage	Bit 19
			0 = System storage not used	
			1 = System storage used	
40	28		Reserved	Bits 20-31
44	2C	Recovery options	Char(4)	
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
57	39	Reserved	Char(7)	
64	40	Context	System pointer	
80	50	Access group	System pointer	
96	60	Length of timestamp in prefix	Bin(2)	
98	62	Length of process name in prefix	Bin(2)	
100	64	Length of user profile name in prefix	Bin(2)	
102	66	Length of program name in prefix	Bin(2)	
104	68	Number of journal spaces attached to the journal port	Bin(2)	
106	6A	Journal entry force count	Bin(4)	
110	6E	Journal port flags	Char(1)	
110	6E		Default journal port	Bit 0
			0 = Not a default journal port for implicitly journaled objects.	
			1 = Default journal port for implicitly journaled objects.	
110	6E		Journal port commit quiesce status	Bit 1
			0 = No quiesce of all commit activity to a transaction boundary is in progress against this journal port.	
			1 = The quiesce of all commit activity to a transaction boundary is in progress against this journal port.	
110	6E		Discard transient entries	Bit 2
			0 = Do not discard transient entries in the permanent journal space.	
			1 = Discard transient entries in the permanent journal space.	
110	6E		Remote journal port	Bit 3
			0 = Not a remote journal port	
			1 = Remote journal port	
110	6E		Obsolete	Bit 4
110	6E		Prevent objects from being journaled	Bit 5
			0 = Objects are allowed to be journaled to this journal port	
			1 = Objects are prevented from being journaled to this journal port	
110	6E		Standby mode	Bit 6

Offset Dec	Offset Hex	Field Name	Data Type and Length		
			0 = Port is not in standby mode		
			1 = Port is in standby mode		
110	6E		Caching mode	Bit 7	
			0 = Port is not in caching mode		
			1 = Port is in caching mode		
111	6F	Default journal port ID	Char(2)		
113	71	Remote journal attributes	Char(3)		
113	71		Receiving environment attribute flags	Char(1)	
113	71		Remote journal environment		Bit 0
			0 = A remote journal receiving environment does not exist for this journal port		
			1 = A remote journal receiving environment does exist for this journal port		
113	71		Delivery mode		Bit 1
			0 = Synchronous delivery mode		
			1 = Asynchronous delivery mode		
113	71		Reserved (binary 0)		Bits 2-7
114	72		Transport mechanism	Char(1)	
			Hex 01 = Bus transport		
			Hex 02 = SNA		
			Hex 03 = TCP/IP		
115	73		Environment status	Char(1)	
			Hex 00 = Unknown		
			Hex 01 = Active		
			Hex 02 = Catch-up in progress		
			Hex 03 = Controlled end in progress		
			Hex 04 = Suspended		
			Hex 05 = Error		
116	74	Number of remote sending environments	UBin(4)		
120	78	Number of journaled objects	UBin(4)		
124	7C	Fixed length data	Char(1)		
124	7C		Include program context name and ASP number	Bit 0	
			0 = Program context name and ASP number will not be in journal entries.		
			1 = Program context name and ASP number will be in journal entries.		
124	7C		Include system sequence number	Bit 1	
			0 = System sequence number will not be in journal entries.		
			1 = System sequence number will be in journal entries.		

Offset Dec	Hex	Field Name	Data Type and Length	
124	7C		Include remote address	Bit 2
			0 = Remote address will not be in journal entries.	
			1 = Remote address will be in journal entries.	
124	7C		Include thread ID	Bit 3
			0 = Thread identifier will not be in journal entries.	
			1 = Thread identifier will be in journal entries.	
124	7C		Include logical unit of work	Bit 4
			0 = Logical unit of work will not be in journal entries.	
			1 = Logical unit of work will be in journal entries.	
124	7C		Include transaction identifier	Bit 5
			0 = Transaction identifier will not be in journal entries.	
			1 = Transaction identifier will be in journal entries.	
124	7C		Reserved (binary 0)	Bits 6-7
125	7D	Quiesced Status	Char(1)	
		Hex 00 = Journal port is not currently quiesced		
		'S'= Journal port has been quiesced such that there are no current open commit cycles and no additional cycles are allowed to start (this is the quiesced state achieved in support of traditional Save While Active)		
		'R'= Journal port has been quiesced only to the point where no operations are in flux. This does not assure that commit cycles are closed. (this is the quiesced state achieved in support of Ragged Save While Active)		
126	7E	Reserved (binary 0)	Char(2)	
128	80	Journal spaces (0 to n)	[*] System pointer	
*	*	Sending environment templates (0 to m)	[*] Char(48)	
*	*	Minimal entry array	Char(32)	
*	*	—- End —-		

The *receiver* must be aligned on a 16-byte boundary. The first 4 bytes of the *receiver* identify the **total number of bytes provided** by the user for the materialization and the next 4 specify the **total number of bytes available** to be materialized. If fewer than 8 bytes are available in the space identified by the *receiver*, operand 1, a *materialization length invalid* (hex 3803) exception is signaled. The instruction materializes as many bytes as can be contained in the receiver's space. If the space of the *receiver* is greater than that required to contain the information requested for materialization, the excess bytes are unchanged. No exceptions are signaled in the event that the *receiver* contains insufficient space for the materialization other than the *materialization length invalid* (hex 3803) exception described previously.

Each journal space currently attached to the journal port will be identified in the **journal space** list. This list has as many entries as are identified in the **number of journal spaces attached to the journal port** field.

The **journal entry force count** is the value which is used to determine how many journal entries are allowed to be deposited on the journal before forcing those entries to disk.

The **discard transient entries** value indicates whether entries of a transient nature, those used strictly for the recovery of objects at IPL time, should be permanently associated with the journal space.

The **remote journal port** field indicates whether this is a remote journal port. A remote journal port receives entries from another journal port via a remote journal receiving environment. No entries can be deposited to a remote journal port.

The **prevent entries from being deposited** field indicates whether journal entries are currently allowed to be deposited to the journal port. Any attempt to deposit other entries to a journal with this attribute in effect will result in an *entry not journaled* (hex 3002) exception being signaled. This field does not apply to remote journal ports and will contain binary 0 if the *remote journal port* field contains a value of binary 1.

The **default journal port ID** will be zeros if the journal port is not a default journal.

The **prevent objects from being journaled** field indicates whether objects are allowed to be journaled to this journal port.

The **standby mode** field indicates whether the port is currently in standby mode. Those ports which are in standby mode allow only a small select subset of critical journal entries to be deposited into the journal space.

The **caching mode** field indicates whether the port is currently in caching mode. Those ports which are in caching mode do not immediately write all journal entries to disk. Rather, they allow most journal entries to linger within a main memory cache until the cache is nearly full.

The **remote journal attributes** fields indicate the status of the receiving environment for remote journal ports. The values in these fields only apply to *remote journal ports* and will be binary 0's for non-remote journal ports.

The **remote journal environment** field indicates whether a remote journal receiving environment exists for this journal port.

The **delivery mode** field indicates how journal entries are sent to this journal port. A *synchronous* delivery mode means that journal entries are sent to this journal concurrently with the entry being deposited on the journal port on the source system. An *asynchronous* delivery mode means that journal entries are sent to this journal port at some time after entries have been deposited on the journal port on the source system. The *delivery mode* does not apply and will contain a value of binary 0 if a remote journal receiving environment does not exist for the journal port or the *environment status* field indicates that the environment is in a suspended or unknown state.

The **transport mechanism** field indicates whether bus-level support or communications-level support is being used to transport the journal entries. The *transport mechanism* field in the *remote journal attributes* section of the materialize template indicates the mechanism used by this remote journal port to receive entries from another source journal port. The *transport mechanism* field does not apply and will contain a value of binary 0 if a remote journal receiving environment does not exist for the journal port. The *transport mechanism* field in the *sending environment template(s)* indicates the mechanism used to send journal entries from this journal port to the remote journal port.

The **environment status** field indicates the current state of the remote journal environment. A value of hex 01 indicates an active remote journaling environment. A value of hex 02 indicates a remote journal port is currently in the process of catching up journal entries from a source journal port. The catch-up phase is considered complete when the transition is made to either synchronous or asynchronous remote journaling. A value of hex 02 in the *environment status* field in the *remote journal attributes* section of the materialize template indicates this remote journal port is currently catching up entries from another source journal port. A value of hex 02 in the *environment status* field in the *sending environment template(s)* indicates the remote journal port is currently in the process of catching up journal entries from this journal port. A value of hex 03 indicates that the remote environment is ending controlled. A value of hex 04 indicates that the environment has been suspended because the environment did not successfully transition to an active state following the catch-up phase. A value of hex 05 indicates that an error has been detected and the environment is waiting to be deleted. The *environment status* field in the *remote journaling attributes* section of the materialize template does not apply and will have a value of binary 0 if a remote journal receiving environment does not exist for the journal port.

The **number of remote sending environments** field indicates the number of remote journal ports that this journal port is actively sending journal entries to. Information about each of the remote journal ports is provided in the *sending environment template(s)* below.

The **number of journaled objects** field indicates the number of objects actively being journaled to this port. This includes both implicitly and explicitly journaled objects.

The **fixed length data** field indicates whether various fixed length data is in the journal entries. A value of binary 1 for any of the following fields indicates that that specific data will be included in each journal entry in the journal space attached to the journal port.

- **include logical unit of work**
- **include transaction identifier**
- **include remote address**
- **include program context name and ASP number**
- **include system sequence number**
- **include thread identifier**

A value of binary 0 indicates that that specific data will NOT be included in those journal entries.

The **minimal entry array** field is defined as an array of bits, numbered from 0 to 255, one bit per entry type. Each bit indicates that minimal journal entries may or may not be deposited to this port for the corresponding entry type. For an entry type N, *minimal entry by entry type* bit N = 0 indicates that minimal entries are NOT accepted for the entry type N. *Minimal entry by entry type* bit N = 1 indicates that minimal entries are accepted for the entry type N.

The **entry type** indicates the type of object which is having its change activity journaled. In most cases, the *entry type* corresponds directly to the object's MI type. However, some MI types are used to represent more than one flavor of object. For this reason, the *entry type* for Data Area's is hex A0. All other objects use the MI type.

The **sending environment template** is repeated for each sending environment. The format of the *sending environment template* is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Sending environment template	Char(48)	
0	0		Remote journal ID	Char(10)
10	A		Sending enviroment attributes	Char(1)
10	A		Delivery mode	Bit 0

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = Synchronous delivery mode	
			1 = Asynchronous delivery mode	
10	A		Reserved (binary 0)	Bits 1-7
11	B		Transport mechanism	Char(1)
			Hex 01 = Bus transport	
			Hex 02 = SNA	
			Hex 03 = TCP/IP	
12	C		Environment status	Char(1)
			Hex 00 = Unknown	
			Hex 01 = Active	
			Hex 02 = Catch-up in progress	
			Hex 03 = Controlled end in progress	
			Hex 04 = Suspended	
			Hex 05 = Error	
13	D		Reserved (binary 0)	Char(3)
16	10		Priority of asynchronous sending task	Char(1)
17	11		Reserved (binary 0)	Char(31)
48	30	—- End —-		

The **priority of asynchronous sending task** field contains the priority of the task sending journal entries to the remote journal port relative to the priority of processes on the machine. This field only applies to asynchronous remote journal sending environments and will contain a value of binary 0 for synchronous remote journal sending environments associated with the journal port.

If operand 2 is a system pointer, it identifies the input *journal port* object. If operand 2 is a space pointer, it provides addressability to the *materialize template*. The *materialize template* is used to identify default journal ports by ASP and default journal port ID. Returned in the *materialize template* is a system pointer to the currently known default journal port with the specified ID. The format of the *materialize template* is as follows:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Materialize template	Char(32)	
0	0		ASP	Char(2) +
2	2		Default journal port ID	Char(2) +
4	4		Reserved (binary 0)	Char(12)
16	10		Journal port	System pointer
32	20	—- End —-		

Note: The fields marked with a plus sign (+) are input to this instruction.

The *materialize template* must be aligned on a 16-byte boundary.

The **ASP field** indicates the ASP on which the default journal port resides.

The **default journal port ID** uniquely identifies the default journal port on the specified ASP.

The journal port currently known as the default journal port of the specified ID and ASP is returned. If there is no known default port, a null pointer value will be returned.

Warning: Temporary Level 3 Header

Authorization Required

•
- Execute
  –
  – Contexts referenced for address resolution
- Operational
  –
  – Operand 2 (if system pointer)

Lock Enforcement

•
- Materialization
  –
  – Operand 2 (if system pointer)
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Journal Space Attributes (MATJSAT)

Op Code (Hex)	Operand 1	Operand 2
05BE	Receiver	Journal space

Operand 1: Space pointer.

Operand 2: System pointer.

Bound program access

```
Built-in number for MATJSAT is 85.
MATJSAT (
         receiver        : address
         journal_space   : address of system pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction materializes the current attributes of the *journal space* specified by operand 2 and places the attributes in the *receiver* specified by operand 1.

The format of the materialization data is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization length	Char(8)	
0	0		Number of bytes provided by user	Bin(4)
4	4		Number of bytes available to be materialized	Bin(4)
8	8	Object identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Object creation options	Char(4)	
40	28		Existence attributes (binary 1)	Bit 0
40	28		Primary associated space attributes	Bit 1
			0 = Fixed-length	
			1 = Variable-length	
40	28		Initial context	Bit 2
40	28		Access group	Bit 3
40	28		Replace option	Bit 4
40	28		Reserved	Bits 5-12
40	28		Initialize primary associated space	Bit 13
40	28		Reserved	Bits 14-18
40	28		Use system storage	Bit 19
			0 = System storage not used	
			1 = System storage used	
40	28		Reserved (binary 0)	Bits 20-31
44	2C	Recovery options	Char(4)	
48	30	Size of primary associated space	Bin(4)	
52	34	Initial value of primary associated space	Char(1)	
53	35	Performance class	Char(4)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
57	39	Reserved (binary 0)	Char(7)	
64	40	Context	System pointer	
80	50	Access group	System pointer	
96	60	Narrow maximum threshold (in bytes)	Bin(4)	
100	64	Narrow minimum threshold value (in bytes)	Bin(4)	
104	68	Percent threshold	Bin(2)	
106	6A	Journal space flags	Char(1)	
106	6A		Reserved (binary 0)	Bit 0
106	6A		Default journal space **0 =** Not a default journal space **1 =** Default journal space	Bit 1
106	6A		Reserved	Bits 2-7
107	6B	Journal space capacity value **Hex 00 =** Maximum sequence number: 2,147,483,136 and maximum journal space size: 2 Gigabytes **Hex 01 =** Maximum sequence number: 9,999,999,999 and maximum journal space size: 1 Terabyte **Hex 02 =** Maximum sequence number: 9,999,999,999 and maximum journal space size: 1 Terabyte and Maximum entry size: 4,000,000,000 bytes **Hex 03 =** Maximum sequence number: 18,446,744,073,709,551,600 and maximum journal space size: 1 Terabyte and Maximum entry size: 4,000,000,000 bytes	Char(1)	
108	6C	Reserved (binary 0)	Char(16)	
124	7C	Narrow last confirmed sequence number	Bin(4)	
128	80	Journal port	System pointer	
144	90	Narrow number of journal entries	Bin(4)	
148	94	Narrow first sequence number	Bin(4)	
152	98	Narrow last sequence number	Bin(4)	
156	9C	Generation number	UBin(4)	
160	A0	Time journal space attached to journal port	Char(8)	
168	A8	Time journal space detached from journal port	Char(8)	
176	B0	Length of timestamp	Bin(2)	
178	B2	Length of process name	Bin(2)	
180	B4	Length of user profile name	Bin(2)	
182	B6	Length of program name	Bin(2)	
184	B8	Fixed length data	Char(1)	
184	B8		Include program context name and ASP number **0 =** Program context name and ASP number will not be in journal entries. **1 =** Program context name and ASP number will be in journal entries.	Bit 0
184	B8		Include system sequence number	Bit 1

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = System sequence number will not be in journal entries.	
			1 = System sequence number will be in journal entries.	
184	B8		Include remote address	Bit 2
			0 = Remote address will not be in journal entries.	
			1 = Remote address will be in journal entries.	
184	B8		Include thread ID	Bit 3
			0 = Thread identifier will not be in journal entries.	
			1 = Thread identifier will be in journal entries.	
184	B8		Include logical unit of work	Bit 4
			0 = Logical unit of work will not be in journal entries.	
			1 = Logical unit of work will be in journal entries.	
184	B8		Include transaction identifier	Bit 5
			0 = Transaction identifier will not be in journal entries.	
			1 = Transaction identifier will be in journal entries.	
184	B8		Reserved (binary 0)	Bits 6-7
185	B9	Reserved (binary 0)	Char(3)	
188	BC	Narrow last journal entry dumped	Bin(4)	
192	C0	Journal space status	Char(2)	
192	C0		Operable journal space	Bit 0
			0 = Journal space is operable	
			1 = Journal space is not operable	
192	C0		Missing journal entries	Bit 1
			0 = No entries missing	
			1 = 1 or more entries missing	
192	C0		Journal space size extension	Bit 2
			0 = Journal space could be extended	
			1 = Journal space could not be extended	
192	C0		Maximum sequence number reached	Bit 3
			0 = Maximum sequence number has not been reached	
			1 = Maximum sequence number has been reached	
192	C0		Journal failure	Bit 4
			0 = No journal failure has occurred	
			1 = A journal failure has occurred	
192	C0		Recoverable commit boundary	Bit 5

Offset Dec	Hex	Field Name	Data Type and Length		
			0 =	All objects with changes journaled to this journal space are at a recoverable commit boundary	
			1 =	One or more objects with changes journaled to this journal space are not at a recoverable commit boundary	
192	C0		Journal space is attached		Bit 6
			0 =	Journal space is not attached to a journal port	
			1 =	Journal space is attached to a journal port	
192	C0		Reserved (binary 0)		Bits 7-15
194	C2	Entry specific data longest length table	Char(*)		
194	C2		Number of entries in table		Bin(4)
198	C6		Entry specific data length elements		[*] Char(6)
198	C6		Entry specific data ID		Char(
200	C8		Entry specific data longest length		UBin(
*	*		Maximum threshold (in basic storage units)		Bin(4)
*	*		Minimum threshold value (in basic storage units)		Bin(4)
*	*		Last confirmed sequence number		Char(8)
*	*		Number of journal entries		Char(8)
*	*		First sequence number		Char(8)
*	*		Last sequence number		Char(8)
*	*		Last journal entry dumped		Char(8)
*	*		Minimal entry array by entry type		Char(32)
*	*	—- End —-			

The *receiver* must be aligned on a 16-byte boundary. The first 4 bytes of the *receiver* identify the **total number of bytes provided** by the user for the materialization and the next 4 specify on output the **total number of bytes available** to be materialized. If fewer than 8 bytes are available in the space identified by the *receiver* (operand 1), a *materialization length invalid* (hex 3803) exception is signaled. The instruction materializes as many bytes as can be contained in the receiver's space. If the space of the *receiver* is greater than that required to contain the information requested for materialization, the excess bytes are unchanged. No exceptions are signaled in the event that the *receiver* contains insufficient space for the materialization other than the *materialization length invalid* (hex 3803) exception described previously.

The **journal space capacity value** field returns the upper limits that are being imposed on both the maximum size of this journal space, the maximum journal sequence number allowed, and the maximum size for a journal entry.

The **last confirmed sequence number** field contains the journal sequence number of the last valid journal entry contained in this *journal space*. If the *number of journal entries* field contains a value of 0, this field will also contain a value of 0.

The **journal port** field contains a system pointer to the journal port to which the designated *journal space* is currently attached. If the *journal space* is not currently attached to a journal port, this field will contain binary 0's.

The **number of journal entries** field contains the number of journal entries currently in the journal space. If this field contains a value of 0, the *journal space* has never been attached to a journal port.

The **first sequence number** field contains the journal sequence number of the first journal entry contained in this *journal space*. If the *number of journal entries* field contains a value of 0, this field will also contain a value of 0.

The **last sequence number** field contains the journal sequence number of the last journal entry contained in this *journal space*. If the *number of journal entries* field contains a value of 0, this field will also contain a value of 0. If journal entries are cloaked in the *journal space*, this field will be the sequence number of the last uncloaked journal entry.

The **generation number** field contains the count of the number of times the sequence number had been reset at the time the *journal space* was attached.

The **time journal space attached to journal port** field contains a timestamp that indicates the time the journal space was attached to a journal port. If the journal space has never been attached to a journal port, a value of 0 will be returned in this field.

The **time journal space detached from journal port** field contains a timestamp that indicates the time the *journal space* was detached from a journal port. If the *journal space* has never been attached to a journal port or is currently attached to a journal port, a value of 0 will be returned in this field.

The **length of timestamp** field contains the length of the timestamp field in the journal prefix of journal entries contained on the *journal space*. If the *journal space* has never been attached to a journal receiver (*number of journal entries* is equal to 0), this field will contain a value of 0.

The **length of process name** field contains the length of the process name field in the journal prefix of journal entries contained on the *journal space*. If the *journal space* has never been attached to a journal receiver (*number of journal entries* is equal to 0), this field will contain a value of 0.

The **length of user profile name** field contains the length of the user profile name field in the journal prefix of journal entries contained on the *journal space*. If the *journal space* has never been attached to a journal receiver (*number of journal entries* is equal to 0), this field will contain a value of 0.

The **length of program name** field contains the length of the program name field in the journal prefix of journal entries contained on the *journal space*. If the *journal space* has never been attached to a journal receiver (*number of journal entries* is equal to 0), this field will contain a value of 0.

The **fixed length data** field indicates whether various fixed length data is in the journal entries. A value of binary 1 for the following fields indicates that that specific data will be included in each journal entry in the journal space attached to the journal port.

- **include logical unit of work**
- **include transaction identifier**
- **include remote address**
- **include program context name and ASP number**
- **include system sequence number**
- **include thread identifier**

A value of binary 0 indicates that that specific data will NOT be included in those journal entries.

The **last journal entry dumped** field contains the journal sequence number of the last complete journal entry that has been dumped from this *journal space*. If no dump operation has been performed on this *journal space*, a value of 0 will be returned.

The **journal space status** fields indicate whether or not the journal space is currently actively receiving journal entries or successfully received all journal entries while it was attached to a journal port. These

fields also indicate the reason journal entries were not placed on the journal space and whether all objects with changes journaled to this journal space are currently at a recoverable commit boundary.

The **operable journal space** field indicates whether or not journal entries are being placed in the *journal space* while it is attached to the indicated journal port. If the *journal space* is no longer attached to a journal port, this field indicates the status of the *journal space* when it was detached from the journal port.

The **missing journal entries** field indicates whether or not journal entries have been created, while this *journal space* was attached to a journal port, that were not recorded on this *journal space*.

The **journal space size extension** field indicates whether or not the *journal space* can be extended. A value of binary 1 in this field indicates a *user profile storage limit exceeded* (hex 2E01) exception was encountered while trying to extend the *journal space*.

The **journal failure** field indicates whether or not a journal failure occurred while this *journal space* was attached to a journal port.

The **recoverable commit boundary** field indicates whether all objects with changes journaled to this journal space are at a recoverable commit boundary. A value of binary 0 in this field indicates that all objects are at a recoverable commit boundary.

The **minimal entry array by entry type** field is defined as an array of bits, numbered from 0 to 255, one bit per entry type. Each bit indicates that minimal journal entries may or may not have been deposited to this space for the corresponding entry type. For an entry type N, *minimal entry array by entry type* bit N = 0 indicates that minimal entries do NOT exist within the space for the entry type N. *Minimal entry array by entry type* bit N = 1 indicates that minimal entries might exist within the space for the entry type N. Entry types are defined in the MATJPAT instruction.

The **entry specific data longest length** table contains the longest length associated with each entry specific data ID found on the *journal space*. If the *journal space* has never been attached to a journal port or there are no entries on the *journal space*, a value of 0 will be returned in the *number of entries in table* field.

Note:

A number of fields (*last confirmed sequence number,number of journal entries,first sequence number,last sequence number,last journal entry dumped*) deliberately appear twice in the materialization data, once in a 'Narrow' representation and again in a wider (UBin 8) representation. When the value returned for each such field is less than 2G (2,147,483,648) it will be returned in both the wide and narrow representation. However, when the actual value is too large to be represented in a Bin(4) field, the narrow instance of such a field will contain, instead, the value of -1. In a similar fashion, both the *maximum threshold* and *minimum threshold value* fields appear twice. The 'Narrow' designated representation is the traditional byte count while the ordinary representation is expressed in basic storage units. For a small capacity journal space both fields will contain proper values in their respective units. However, for a large capacity (i.e. 1 Terabyte) journal space the thresholds may be too large to be represented as a byte count. In that instance the 'Narrow' representations will each contain the value of -1 and the true threshold sizes will be present only in the ordinary threshold fields.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

–
– Contexts referenced for address resolution
- Operational
  –
  – Operand 2

Lock Enforcement

•
- Materialization
  –
  – Operand 2
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Machine Attributes (MATMATR)

Op Code (Hex)	Operand 1	Operand 2
0636	Materialization	Machine attributes

Operand 1: Space pointer.

Operand 2: Character(2) scalar or space pointer.

Bound program access
Built-in number for MATMATR1 is 92. MATMATR1 (materialization : address machine_attributes : address (of just a selector value))

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The instruction makes available the unique values of machine attributes. Unless otherwise stated, all options materialize the value of machine attributes for the current partition. The values of various machine attributes are placed in the receiver.

Operand 2 specifies options for the type of information to be materialized. Operand 2 is specified as an attribute selection value (character(2) scalar) or as an attribute selection template (space pointer to a character(2) scalar). The machine attributes are divided into nine groups. Byte 0 of the attribute selection operand specifies from which group the machine attributes are to be materialized. Byte 1 of the options operand selects a specific subset of that group of machine attributes.

Operand 1 specifies a space pointer to the area where the materialization is to be placed. The format of the materialization is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided by the user	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Attribute specification	Char(*)	
		(as defined by the attribute selection)		
*	*	—- End —-		

The first 4 bytes of the materialization (operand 1) identify the total **number of bytes provided by the user** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available for materialization**. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested for materialization, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

Table 1. MATMATR Selection Values

Selection value	Attribute Description	Page
0004	Machine serial identification	reference #1 (page)
0100	Time-of-day clock (local time)	reference #2 (page)
0101	Time-of-day clock with clock-offset	reference #3 (page)
0104	Primary initial process definition template	reference #4 (page)
0108	Machine initialization status record	reference #5 (page)

Selection value	Attribute Description	Page
0118	Uninterruptible power supply delay time and calculated delay time	reference #6 (page)
012C	Vital product data	reference #7 (page)
0130	Network attributes	reference #8 (page)
0134	Date format	reference #9 (page)
0138	Leap year adjustment	reference #10 (page)
013C	Timed power on	reference #11 (page)
0140	Timed power on enable/disable	reference #12 (page)
0144	Remote power on enable/disable	reference #13 (page)
0148	Auto power restart enable/disable	reference #14 (page)
014C	Date separator	reference #15 (page)
0151	System security indicators	reference #16 (page)
0161	Perform hardware checks on IPL	reference #17 (page)
0164	Uninterruptible power supply type	reference #18 (page)
0168	Panel status request	reference #19 (page)
016C	Extended machine initialization status record	reference #20 (page)
0170	Alternate initial process definition template	reference #21 (page)
0178	Hardware storage protection enforcement state	reference #22 (page)
0180	Time separator	reference #23 (page)
0184	Software error logging	reference #24 (page)
0188	Machine task or secondary thread termination event control option	reference #25 (page)
01A8	Service attributes	reference #26 (page)
01B0	Signal controls	reference #27 (page)
01C8	Cryptography attributes	reference #28 (page)
01D0	Communication network attributes	reference #29 (page)
01DC	Installed processor count	reference #30 (page)
01E0	Partitioning information	reference #31 (page)
01EC	Additional load source reserved space	reference #32 (page)
01F4	Processor on demand information	reference #33 (page)
01F6	Memory on demand information	reference #34 (page)
01F8	IPL identifier	reference #36 (page)
01FC	Electronic licensing identifier	reference #37 (page)
0200	Wait state performance information	reference #38 (page)
0204	Hardware Management Console (HMC) information	reference #39 (page)

The machine attributes selected by operand 2 are materialized according to the following selection values:

Selection Value
Hex 0004

Attribute Description *(Ref #1.)*
Machine serial identification

The machine serial identification that is materialized is an 8-byte character field that contains the unique physical machine identifier. This identifier is the same for all partitions of a physical machine. *(Ref #2.)*

Hex 0100

Time-of-day clock (local time)

The time-of-day clock option is used to return the time-of-day clock as the local time for the system. The MATMDATA or MATTODAT instruction can be used to return the time-of-day clock as the Coordinated Universal TIme (UTC) for the system. See "Standard Time Format" on page 1272 for additional information on the time-of-day clock.

The maximum unsigned binary value that the time of day clock can be modified to contain is hex DFFFFFFFFFFFFFFF. *(Ref #3.)*

Hex 0101

Time-of-day clock with clock-offset

In addition to returning the system time-of-day (TOD) clock (as defined for selection value hex 0100 described previously), the *time-of-day clock with clock-offset* option will also return a **clock-offset** which can be used to convert clocks from different partitions (on a partitioned system) to values referencing the common hardware clock used by all partitions on the same system. This enables users/administrators of a partitioned system to analyze and compare events taking place in different partitions. See "Standard Time Format" on page 1272 for additional information on the time-of-day clock.

The *clock-offset* is defined as the difference between the time-of-day clock for the current partition and the value of the hardware clock counter.

The materialize format of the *time-of-day clock with clock-offset* is as follows:

Offset

Dec	Hex
	Field Name
	Data Type and Length
8	8
	Time-of-day clock
	Char(8)
16	10
	Clock-offset
	Char(8)
24	18
	—- End —-

(Ref #4.)

Hex 0104

Primary initial process definition template

The primary initial process definition template is used by the machine to perform an initial program load.

No check is made and no exception is signaled if the values in the template are invalid; however, the next initial program load will not be successful. *(Ref #5.)*

Hex 0108

Machine initialization status record

The MISR (machine initialization status record) is used to report the status of the machine. The status is initially collected at IPL and then updated as system status changes.

The materialize format of the MISR is as follows:

Offset		
Dec	Hex	Field Name / Data Type and Length
8	8	MISR status Char(6)
8	8	Restart IMPL Bit 0 **0 =** IMPL was not initiated by the Terminate instruction **1 =** IMPL was initiated by the Terminate instruction
8	8	Manual power on Bit 1 **0 =** Power on not due to Manual power on **1 =** Manual power on occurred
8	8	Timed power on Bit 2 **0 =** Power on not due to Timed power on **1 =** Timed power on occurred
8	8	Remote power on Bit 3 **0 =** Power on not due to remote power on **1 =** Remote power on occurred
8	8	

Hex 0118

Uninterruptible power supply delay time and calculated delay time. Note: The UPS delay time is meaningful only if a UPS is installed.

The format of the template for the uninterruptible power supply delay time (including the 8-byte prefix) is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	UPS Delay time
	Bin(4)
12	C
	Calculated UPS Delay time
	Bin(4)
	The delay time interval is the amount of time the system waits for the return of utility power. If a utility power failure occurs, the system will continue operating on the UPS supplied power. If utility power does not return within the user specified delay time, the system will perform a quick power down. The delay time interval is set by the customer. The calculated delay time is determined by the amount of main storage and DASD that exists on the system. Both values are in seconds.
16	10
	—- End —-

(Ref #7.)

Hex 012C

Vital product data

The VPD (vital product data) is a template that contains information for memory card VPD, processor VPD, Columbia/Colomis VPD, central electronic complex (CEC) VPD and the panel VPD. The VPD information that is materialized is the same for all partitions of a physical machine.

The materialize format of the VPD (Including the 8-byte prefix) is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Reserved
	Char(8)
16	10
	System VPD location
	Char(32)
16	10
	Offset to memory VPD
	Bin(4)
20	14
	Offset to processor VPD
	Bin(4)
24	18
	Offset to Columbia/Colomis
	Bin(4)
28	1C
	Offset to CEC VPD
	Bin(4)
32	20
	Offset to panel VPD
	Bin(4)
36	24
	Reserved
	Char(12)
48	30
	Main store memory VPD
	Char(1040)
48	30
	Usable memory installed
	Bin(2)
	(In megabytes)
50	32
	Minimum memory required

Hex 0130

Network attributes

The network attributes is a template that contains information concerning APPN network attributes.

The materialize format of the network attributes is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Network data
	Char(190)
8	8
	System name
	Char(8)
16	10
	System name length
	Bin(2)
18	12
	New system name
	Char(8)
26	1A
	New system name length
	Bin(2)
28	1C
	Local system network identification
	Char(8)
36	24
	Local system network identification length
	Bin(2)
38	26
	End node data compression
	Bin(4)
42	2A
	Intermediate node data compression
	Bin(4)
46	2E
	Reserved
	Char(2)
48	30
	Local system control point name
	Char(8)
56	38
	Local system control point name length
	Bin(2)
58	3A
	Maximum APPN LUDs on virtual APPN CDs

Hex 0134

Date format

The date format is the format in which the date will be presented to the customer. The possible values are YMD, MDY, DMY where Y = Year, M = Month, D = Day and JUL = Julian.

The format of the template for date format is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Date format
	Char(3)
11	B
	—- End —-

(Ref #10.)

Hex 0138

Leap year adjustment

The leap year adjustment is added to the leap year calculations to determine the year in which the leap should occur. The valid values are 0, 1, 2, 3.

The format of the template for leap year adjustment is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Leap year adjustment
	Bin(2)
10	A
	—- End —-

(Ref #11.)

Hex 013C

Timed power on

The timed power on is the time and date at which the system should automatically power on if it is not already powered on. If the physical machine is powered off, and the time and date at which a partition is automatically powered is set to occur before the time and date at which the physical machine is set to automatically power on (according to their respective time of day clocks), then the partition will be powered on when the physical machine is powered on.

The format of the template for timed power on is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Minute
	Bin(2)
10	A
	Hour
	Bin(2)
12	C
	Day
	Bin(2)
14	E
	Month
	Bin(2)
16	10
	Year
	Bin(2)
18	12
	—- End —-

(Ref #12.)

Hex 0140

Timed power on enable/disable

The timed power on enable/disable allows the timed power on function to be queried to determine if the function is enabled or disabled.

The format of the template for timed power on enable/disable is as follows:

Offset

Dec	Hex
	Field Name
	Data Type and Length
8	8
	Enable/disable
	Bin(2)
	Hex 8000 = Timed power on is enabled
	Hex 0000 = Timed power on is disabled
10	A
	—- End —-

(Ref #13.)

Hex 0144

Remote power on enable/disable

The remote power on enable/disable allows the remote power on function to be queried to determine if the function is enabled or disabled.

The format of the template for remote power on enable/disable is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Enable/disable
	Bin(2)
	Hex 8000 = Remote power on is enabled
	Hex 0000 = Remote power on is disabled
10	A
	—- End —-

(Ref #14.)

Hex 0148

Auto power restart enable/disable

The auto power restart enable/disable allows the auto power restart function to be queried to determine if the function is enabled or disabled.

The format of the template for auto power restart enable/disable is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Enable/disable
	Bin(2)
	Hex 8000 = Auto power restart is enabled
	Hex 0000 = Auto power restart is disabled
10	A
	—- End —-

(Ref #15.)

Hex 014C

Date separator

The date separator is used when the date is presented to the customer. The valid values are a slash(/), dash(-), period(.), comma(,) and a blank( ).

The format of the template for the date separator is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Date separator
	Char(1)
9	9
	—- End —-

(Ref #16.)

Hex 0151

System security indicators (Can only be materialized)

The *system security indicators* return the current setting of the system security flags.

The format of the template for *system security indicators* is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	System security indicators
	Char(1)
8	8
	Reserved
	Bits 0-4
8	8
	Restrict change of service tool user ID passwords by operating system
	Bit 5
	0 = Not restricted
	1 = Restricted
8	8
	Restrict change of operating system security values
	Bit 6
	0 = Not restricted
	1 = Restricted
8	8
	Restrict adds to digital certificates store
	Bit 7
	0 = Not restricted
	1 = Restricted
9	9
	Reserved (binary 0)
	Char(11)
20	14
	—- End —-

Hex 0161

Perform hardware checks on IPL

The perform hardware checks on IPL option retrieves the current setting that indicates if the system is skipping hardware checks on all physical machine IPLs. The IPL checks are performed only when the physical machine is IPLed.

The format of the template for perform hardware checks is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Perform/not perform
	Bin(2)
	Hex 8000 = The system is doing hardware checks on IPLs.
	Hex 0000 = The system is not checking the hardware on IPLs.
10	A
	—- End —-

(Ref #18.)

Hex 0164

Uninterruptible power supply type

Note: The *UPS type* is meaningful only if a UPS is installed.

The uninterruptible power supply type option allows the MI user to tell the machine how much of the system is powered by a UPS (ie, what type of UPS is installed). A *full UPS* will power all racks in the system. A *limited UPS* will have enough power to perform main store dump. A *mini UPS* will power the racks containing the CEC and the load source.

The format of the template for UPS Type is as follows:

Offset

Dec Hex

Field Name

Data Type and Length

8 8

UPS type

Bin(2)

Hex 0000 =
Indicates a full UPS is installed (all racks have a UPS installed)

Hex 4000 =
Indicates a limited UPS is installed (the UPS only has enough power to do a main store dump)

Hex 8000 =
Indicates a mini UPS is installed (only the minimum number of racks are powered)

10 A

—- End —-

(Ref #19.)

Hex 0168

Panel status request

The panel status request option returns the current status of the operations panel.

The format of the template for panel status request is as follows (including the usual 8-byte prefix):

Offset

Dec	Hex	Field Name	Data Type and Length
8	8	Current IPL type	Char(1)
9	9	Panel status	Char(2)
9	9	Uninterrupted power supply installed	Bit 0

0 = UPS not installed

1 = UPS installed, ready for use

9	9	Utility power failed, running on UPS	Bit 1

0 = Running on utility power

1 = Running on UPS

9	9	Uninterrupted power supply (UPS) bypass active	Bit 2

0 = UPS bypass not active

1 = UPS bypass active

9	9	Uninterrupted power supply (UPS) battery low	Bit 3

0 = UPS battery not low

1 = UPS battery low

Hex 016C

Extended machine initialization status record

The XMISR (extended machine initialization status record) is used to report the status of the machine.

The materialize format of the XMISR is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Save storage status
	Char(4)
8	8
	Reserved (binary 0)
	Bit 0
8	8
	Completion status
	Bit 1

0 = Save storage did not complete

1 = Save storage completed

8	8
	System restored status
	Bit 2

0 = Save storage did not restore the system

1 = Save storage restored the system

8	8
	Save storage attempted
	Bit 3

0 = Save storage not attempted

1 = Save storage was attempted

8	8
	Unreadable sectors
	Bit 4

0 = Unreadable sectors were not found

1 = Unreadable sectors were found during save operation

Hex 0170

Alternate initial process definition template

The alternate initial process definition template is used by the machine when performing an automatic install.

No check is made and no exception is signaled if the values in the template are invalid; however, the next automatic install will not be successful. *(Ref #22.)*

Hex 0178

Hardware storage protection enforcement state

Note: Hardware storage protection is meaningful only on version 2 hardware or later. Hardware storage protection is not supported at all on version 1 hardware.

The hardware storage protection (HSP) mechanism is always in effect. However, HSP is enforced for individual storage areas in two different ways. For some storage areas, HSP is always enforced. For others, HSP is enforced only when this machine attribute is active. Attempted use of any storage area in a manner inconsistent with its storage protection attributes will result in *object domain or hardware storage protection violation* (hex 4401) exception when HSP is being enforced for that storage.

System objects for which HSP is always enforced are:

- programs (object type hex 02)
- modules (object type hex 03)
- XOM objects (object type hex 20)
- any objects with type values greater than hex 20.

HSP is also always enforced for secondary associated spaces. In addition, some individual objects of type space or index, and the primary associated spaces of all MI objects, can optionally be protected with HSP enforcement at all times.

The format of the template for the *hardware storage protection enforcement state* option is as follows (including the usual 8-byte prefix):

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Hardware storage protection enforcement state
	Bin(2)
	Hex 0000 = Indicates hardware storage protection is enforced only for storage that is always protected
	Hex 8000 = Indicates hardware storage protection is enforced for all storage
10	A
	—- End —-

(Ref #23.)

Hex 0180

Time separator

The time separator is used when the time is presented to the customer. The valid values are a colon(:), period(.), comma(,) and a blank( ).

The format of the template for the time separator is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Time separator
	Char(1)
9	9
	—- End —-

(Ref #24.)

Hex 0184

Software error logging

The software error logging machine attribute is used to allow the MI user to determine whether or not software error logging is active for the machine

The format of the template for software error logging is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Software error logging
	Bin(2)
	Hex 8000 = Software error logging is active
	Hex 0000 = Software error logging is not active
10	A
	—- End —-

(Ref #25.)

Hex 0188

Machine task or secondary thread termination event control option

The machine task or secondary thread termination event option controls whether the machine will signal events when machine tasks or secondary threads terminate. The default, which is established every IPL, is to signal neither machine task nor secondary thread termination events.

There are different events associated with the termination of machine tasks and secondary threads. The machine task or secondary thread termination event option is a bit mask, with individual bits corresponding to machine tasks or secondary threads, and their associated events. If a bit is binary 1, the corresponding event will be signalled; if binary 0, it will not.

The events are signalled to the process containing the thread which most recently executed Modify Machine Attributes (MODMATR), specifying the machine task or secondary thread termination event control option attribute selection. If a process terminates while it is the process to which the machine task or secondary thread termination events are to be signalled, the signalling of these events is stopped.

The format of the template for the machine task or secondary thread termination event option is as follows:

Offset

Dec	Hex
	Field Name
	Data Type and Length
8	8
	Machine task or secondary thread termination event option
	Char(2)
8	8
	Signal machine task termination events
	Bit 0
8	8
	Signal secondary thread termination events
	Bit 1
8	8
	Reserved (binary 0)
	Bits 2-15
10	A
	—- End —-

(Ref #26.)

Hex 01A8

Service attributes

The service attributes is a template that contains system serviceability information.

The materialize format of the service attributes (including the 8-byte prefix) is as follows:

Offset

Dec Hex

Field Name

Data Type and Length

8 8

Service attribute flags

Char(1)

8 8

Automatic problem analysis

Bit 0

0 = Automatic problem analysis is not enabled

1 = Automatic problem analysis is enabled

8 8

Automatic problem notification

Bit 1

0 = Automatic problem notification is not enabled

1 = Automatic problem notification is enabled

8 8

Service attributes status

Bit 2

0 = Service attribute values are not set

1 = Service attribute values are set

8 8

Allow remote service access

Bit 3

0 = Remote service access is not allowed

1 = Remote service access is allowed

8 8

Signal controls

The materialization format of the Signal Controls (including the 8-byte prefix) is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Reserved
	Char(8)
16	10
	Signal blocking mask
	Char(8)
16	10
	Reserved (binary 0)
	Bit 0
16	10
	Blocked/unblocked option
	Bits 1-63

0 = Signal is blocked. Signal action for the signal monitor is to be deferred.

1 = Signal is unblocked. Signal action for the signal monitor is eligible to be scheduled.

24	18
	Number of signal monitors
	Bin(4)
28	1C
	Reserved (binary 0)
	Char(4)
32	20
	Signal monitor data
	[*] Char(16)
	(repeated for each signal monitor)
32	20
	Signal number
	Bin(4)
36	24
	Signal action
	Bin(2)

-1 = Signal associated with this signal monitor is

Hex 01C8

Cryptography attributes

The format of the template for cryptography attributes is as follows:

Offset

Dec Hex

Field Name

Data Type and Length

8 8

Number of algorithm entries to follow

UBin(2)

10 A

Algorithm entry

[*] Char(6)

(repeated *number of algorithm entries to follow* times)

10 A

Algorithm identifier

Char(2)

Hex 0001 =
MAC - Message Authentication Code

Hex 0002 =
MD5

Hex 0003 =
SHA-1 - Secure Hash Algorithm

Hex 0004 =
DES (encrypt only) - Data Encryption Standard

Hex 0005 =
DES (encrypt and decrypt)

Hex 0006 =
RC4

Hex 0007 =
RC5

Hex 0008 =
DESX

Hex 0009 =
Triple-DES

Hex 000A =
DSA - Digital Signature Algorithm

Hex 000B =
RSA - Rivest-Shamir-Adleman

Hex 000C =
Diffie-Hellman

Hex 000D =
CDMF - Commercial Data Masking Facility

Hex 000E =
RC2

Hex 000F =
AES - Advanced Encryption Standard

Hex 01D0

Communication network attributes (can be materialized and modified)

The communication network attribute is a template that contains information concerning communication attributes.

The format of the template for the communication network attributes is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Communication attribute
	Char(256)
8	8
	Modem country identifier
	UBin(4)
12	C
	Reserved (binary 0)
	Char(252)
264	108
	—- End —-

The **modem country identifier** specifies the country-specific identifier for modems which are internal to I/O Adapters. This value must be configured correctly to ensure proper operation and, in some countries, to meet legal requirements. There can only be one *modem country identifier* for each partition of a physical machine

The supported *modem country identifiers* are as follows:

Country	Modem Country ID (Hex value)
Argentina	00004152
Aruba	00004157
Australia	00004155
Austria	00004154
Bahrain	00004248
Belgium	00004245
Brazil	00004252
Brunei	0000424E
Canada	00004341
Cayman Islands	00004B59
Chile	0000434C
China	0000434E
Colombia	0000434F
Costa Rica	00004352

Hex 01DC

Installed processor count

This option makes available the installed processor count for the physical machine. The materialization format of installed processor count information (including the 8-byte prefix for *number of bytes provided* and *number of bytes available*) is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Number of installed processors
	UBin(2)
10	A
	—- End —-

Number of installed processors is the number of processors installed on the physical machine.

If the physical machine has the on-demand processors feature, *number of installed processors* = number of permanently activated processors + number of temporarily activated processors + number of processors which are not activated. *(Ref #31.)*

Hex 01E0

Partitioning information

This option makes available partitioning information for the physical machine and the current partition. The materialization format of partitioning information (including the 8-byte prefix for *number of bytes provided* and *number of bytes available*) is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Current number of partitions
	Char(1)
9	9
	Current partition identifier
	Char(1)
10	A
	Primary partition identifier
	Char(1)
11	B
	Service partition identifier
	Char(1)
12	C
	Firmware level
	Char(1)
13	D
	Reserved (binary 0)
	Char(3)
16	10
	Logical serial number
	Char(10)
26	1A
	Reserved (binary 0)
	Char(5)
31	1F
	Partition attributes
	Char(1)
31	1F
	Partition physical processor sharing attribute
	Bit 0

0 = Partition does not share physical processors

1 = Partition shares physical processors

31	1F
	Partition uncapped attribute

Hex 01EC

Additional load source reserved space (Can be modified and materialized)

Use this selection to check if additional load source disk space is allowed to be reserved for system use and if that space has already been reserved. If the indicators show that additional load source disk space is allowed for system use and that LIC has not reserved the space yet, after IPL, LIC will reserve this space and it cannot be freed.

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Reserved disk space indicators
	Char(1)
8	8
	Space reserved indicator
	Bit 0

This indicator indicates what can take effect on the next IPL.

0 = LIC cannot reserve load source disk space.

1 = LIC can reserve load source disk space on the next IPL.

8	8
	Load source space reserved
	Bit 1

0 = LIC has not reserved load source disk space.

1 = LIC has reserved load source disk space.

8	8
	Reserved (binary 0)
	Bits 2-7
9	9
	—- End —-

The **space reserved indicator** field indicates whether additional load source disk space can be reserved by LIC or not on the next IPL. Once the space is reserved by LIC, that is, the value of *load source space reserved* is binary 1, the value of this field no longer has any meaning.

The **load source space reserved** field indicates whether additional load source disk space has been reserved by LIC or not. *(Ref #33.)*

Hex 01F4

On-demand processor information (Can only be materialized)

Use this selection to materialize the information of on-demand processors on the system.

If the system does not have the on-demand processor feature installed, all non-reserved values returned will be blanks (hex 40s) except the enabled and active indicators will be hex 00s, and the *current time of day* will be set. Also, some features may support limited on-demand functions, such as *Capacity Upgrade on Demand* (CUoD) but not *On/Off Capacity on Demand* (CoD) for processors. In such a case, the values returned for the unsupported function will be blanks (hex 40s).

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	System type
	Char(4)
12	C
	System serial number
	Char(10)
22	16
	Capacity card CCIN
	Char(4)
26	1A
	Capacity card serial number
	Char(10)
36	24
	Capacity card unique identifier
	Char(16)
52	34
	Capacity Upgrade on Demand activation feature
	Char(4)
56	38
	Activated Capacity Upgrade on Demand units
	Char(4)
60	3C
	Capacity Upgrade on Demand sequence number
	Char(4)
64	40
	Capacity Upgrade on Demand entry check
	Char(2)
66	42
	Capacity Upgrade on Demand maximum processors that can be purchased
	Char(4)
70	46
	On/Off Capacity on Demand enabled

Hex 01F6

On-demand memory information (Can only be materialized)

Use this selection to materialize the information of on-demand memory on the system.

If the system does not have the on-demand memory feature installed, all non-reserved values returned will be blanks (hex 40s) except the enabled and active indicators will be hex 00s, and the *current time of day* will be set. Also, some features may support limited on-demand functions, such as Capacity Upgrade on Demand but not On/Off Capacity on Demand for memory. In such a case, the values returned for the unsupported function will be blanks (hex 40s).

The format of the template is as follows:

Offset

Dec	Hex	Field Name	Data Type and Length
8	8	System type	Char(4)
12	C	System serial number	Char(10)
22	16	Capacity card CCIN	Char(4)
26	1A	Capacity card serial number	Char(10)
36	24	Capacity card unique identifier	Char(16)
52	34	Capacity Upgrade on Demand activation feature	Char(4)
56	38	Activated Capacity Upgrade on Demand units	Char(4)
60	3C	Capacity Upgrade on Demand sequence number	Char(4)
64	40	Capacity Upgrade on Demand entry check	Char(2)
66	42	Capacity Upgrade on Demand maximum memory that can be purchased	Char(4)
70	46	On/Off Capacity on Demand enabled	

Hex 01F7

On-demand memory information (Can only be materialized) (Internal use only)

This option can only be used in a program running in system state. The *scalar value invalid* (hex 3203) exception will be signaled if the program that issues this option is running in user state.

Use this selection to materialize the information of on-demand memory on the system, including information required to verify resource usage.

If the system does not have the on-demand memory feature installed, all non-reserved values returned will be blanks (hex 40s) except the enabled and active indicators will be hex 00s, and the *current time of day* will be set. Also, some features may support limited on-demand functions, such as Capacity Upgrade on Demand but not On/Off Capacity on Demand for memory. In such a case, the values returned for the unsupported function will be blanks (hex 40s).

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	System type
	Char(4)
12	C
	System serial number
	Char(10)
22	16
	Capacity card CCIN
	Char(4)
26	1A
	Capacity card serial number
	Char(10)
36	24
	Capacity card unique identifier
	Char(16)
52	34
	Capacity Upgrade on Demand activation feature
	Char(4)
56	38
	Activated Capacity Upgrade on Demand units
	Char(4)
60	3C
	Capacity Upgrade on Demand sequence number
	Char(4)
64	40
	Capacity Upgrade on Demand entry check
	Char(2)
66	42

Hex 01F8

IPL identifier (Can only be materialized)

Use this selection to materialize the IPL identifier. This value changes with each IPL.

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	IPL identifier
	UBin(4)
12	C
	—- End —-

The **IPL identifier** is a value that is unique for each system IPL. The value increases for each system IPL. *(Ref #37.)*

Hex 01FC

Electronic licensing identifier (Can be modified and materialized)

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Electronic licensing identifier
	Char(5)
13	D
	—- End —-

The **electronic licensing identifier** field is the value of version, release and modification level of the OS/400 to be installed during the next upgrade whose license is accepted by the customers. The format of the *electronic licensing identifier* is vrmnn where v is the version, r, the release, m, the modication level, and nn are operating system assigned values. *(Ref #38.)*

Hex 0200

Wait state performance information (Can only be materialized)

Use this selection to materialize the wait state performance information.

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Reserved (binary 0)
	Char(8)
16	10
	Offset to descriptor table entries
	UBin(4)
20	14
	Offset to mapping table entries
	UBin(4)
24	18
	Number of descriptor table entries
	UBin(2)
26	1A
	Number of mapping table entries
	UBin(2)
28	1C
	Reserved (binary 0)
	Char(20)
48	30
	Descriptor table entry
	[*] Char(64)
	(repeated *number of descriptor table entries* times)
48	30
	Collection bucket number
	UBin(2)
50	32
	Collection bucket descriptor
	Char(50)
100	64
	Reserved (binary 0)
	Char(12)
*	*
	Mapping table entry
	[*] Char(16)
	(repeated *number of mapping table entries* times)

Hex 0204

Hardware management console information

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Number of entries returned
	UBin(4)
12	C
	Reserved (binary 0)
	Char(4)
16	10
	Hardware Management Console (HMC) information
	[*] Char(1036)
	(repeated for *number of entries returned*)
16	10
	HMC information length
	UBin(2)
18	12
	HMC information
	Char(1034)
*	*
	—- End —-

The **number of entries returned** field returns the number of *Hardware Management Console (HMC) information* entries returned. On a non-HMC managed system, the value returned will be binary 0.

The **HMC information** field returns a string containing the following data:

- HMC name
- HMC host name
- IP address
- HMC state

The data returned is in 7-bit ASCII and its format is as follows: keyword1=its_value;keyword2=its_value;etc. where a keyword can be HscName, HscHostName, HscIPAddr, or HmcStat. Each keyword is followed by an equal sign (=), its value, and ends with a semi-colon (;). For example, an HMC information string can look like this: HscName=679231U*23WW193; HscHostName=hosta.company.xyz.com; HscIPAddr=3.103.123.118;HmcStat=1;

Keywords and their values in the *HMC information* string can be in any order. There is no carriage return <CR> (hex 0D) or line feed <LF> (hex 0A), and the string is not NULL terminated.

The values of HMC state can be

- 1 = the HMC is operating successfully.
- 2 = the HMC has indicated it is temporarily disconnecting.

Hex 0208

Keep current disk configuration during install (can be modified and materialized)

This option is used to materialize the indicator that indicates whether or not all non configured disk units should be added to the system ASP during an automatic install operation.

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Keep current disk configuration indicator
	Char(1)
	Hex 00 = Add all non configured disk units to the system ASP.
	Hex 01 = Keep current disk configuration.
9	9
	Reserved (binary 0)
	Char(7)
16	10
	—- End —-

Limitations (Subject to Change): Data-pointer-defined scalars are not allowed as a primary operand for this instruction. An *invalid operand type* (hex 2A06) exception is signaled if this occurs.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C0A Service Processor Unable to Process Request

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Machine Attributes (MATMATR)

Op Code (Hex)	Operand 1	Operand 2
0636	Materialization	Machine attributes

Operand 1: Space pointer.

Operand 2: Character(2) scalar or space pointer.

Bound program access

```
Built-in number for MATMATR1 is 92.
MATMATR1 (
          materialization      : address
          machine_attributes   : address (of just a selector value)
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The instruction makes available the unique values of machine attributes. Unless otherwise stated, all options materialize the value of machine attributes for the current partition. The values of various machine attributes are placed in the receiver.

Operand 2 specifies options for the type of information to be materialized. Operand 2 is specified as an attribute selection value (character(2) scalar) or as an attribute selection template (space pointer to a character(2) scalar). The machine attributes are divided into nine groups. Byte 0 of the attribute selection operand specifies from which group the machine attributes are to be materialized. Byte 1 of the options operand selects a specific subset of that group of machine attributes.

Operand 1 specifies a space pointer to the area where the materialization is to be placed. The format of the materialization is as follows:

Offset				
Dec	Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided by the user	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Attribute specification	Char(*)	
		(as defined by the attribute selection)		
*	*	—- End —-		

The first 4 bytes of the materialization (operand 1) identify the total **number of bytes provided by the user** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available for materialization**. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested for materialization, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

Table 1. MATMATR Selection Values

Selection value	Attribute Description	Page
0004	Machine serial identification	reference #1 (page)
0100	Time-of-day clock (local time)	reference #2 (page)
0101	Time-of-day clock with clock-offset	reference #3 (page)
0104	Primary initial process definition template	reference #4 (page)
0108	Machine initialization status record	reference #5 (page)
0118	Uninterruptible power supply delay time and calculated delay time	reference #6 (page)
012C	Vital product data	reference #7 (page)
0130	Network attributes	reference #8 (page)
0134	Date format	reference #9 (page)
0138	Leap year adjustment	reference #10 (page)
013C	Timed power on	reference #11 (page)
0140	Timed power on enable/disable	reference #12 (page)
0144	Remote power on enable/disable	reference #13 (page)
0148	Auto power restart enable/disable	reference #14 (page)
014C	Date separator	reference #15 (page)
0151	System security indicators	reference #16 (page)
0161	Perform hardware checks on IPL	reference #17 (page)
0164	Uninterruptible power supply type	reference #18 (page)
0168	Panel status request	reference #19 (page)
016C	Extended machine initialization status record	reference #20 (page)
0170	Alternate initial process definition template	reference #21 (page)
0178	Hardware storage protection enforcement state	reference #22 (page)
0180	Time separator	reference #23 (page)
0184	Software error logging	reference #24 (page)
0188	Machine task or secondary thread termination event control option	reference #25 (page)
01A8	Service attributes	reference #26 (page)
01B0	Signal controls	reference #27 (page)
01C8	Cryptography attributes	reference #28 (page)
01D0	Communication network attributes	reference #29 (page)
01DC	Installed processor count	reference #30 (page)
01E0	Partitioning information	reference #31 (page)
01EC	Additional load source reserved space	reference #32 (page)
01F4	Processor on demand information	reference #33 (page)
01F6	Memory on demand information	reference #34 (page)
01F8	IPL identifier	reference #36 (page)
01FC	Electronic licensing identifier	reference #37 (page)
0200	Wait state performance information	reference #38 (page)

Selection value	Attribute Description	Page
0204	Hardware Management Console (HMC) information	reference #39 (page)

The machine attributes selected by operand 2 are materialized according to the following selection values:

Selection Value — **Attribute Description** *(Ref #1.)*

Hex 0004 — **Machine serial identification**

The machine serial identification that is materialized is an 8-byte character field that contains the unique physical machine identifier. This identifier is the same for all partitions of a physical machine. *(Ref #2.)*

Hex 0100 — **Time-of-day clock (local time)**

The time-of-day clock option is used to return the time-of-day clock as the local time for the system. The MATMDATA or MATTODAT instruction can be used to return the time-of-day clock as the Coordinated Universal TIme (UTC) for the system. See "Standard Time Format" on page 1272 for additional information on the time-of-day clock.

The maximum unsigned binary value that the time of day clock can be modified to contain is hex DFFFFFFFFFFFFFF. *(Ref #3.)*

Hex 0101 — **Time-of-day clock with clock-offset**

In addition to returning the system time-of-day (TOD) clock (as defined for selection value hex 0100 described previously), the *time-of-day clock with clock-offset* option will also return a **clock-offset** which can be used to convert clocks from different partitions (on a partitioned system) to values referencing the common hardware clock used by all partitions on the same system. This enables users/administrators of a partitioned system to analyze and compare events taking place in different partitions. See "Standard Time Format" on page 1272 for additional information on the time-of-day clock.

The *clock-offset* is defined as the difference between the time-of-day clock for the current partition and the value of the hardware clock counter.

The materialize format of the *time-of-day clock with clock-offset* is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Time-of-day clock
	Char(8)
16	10
	Clock-offset
	Char(8)
24	18
	—- End —-

(Ref #4.)

Hex 0104

Primary initial process definition template

The primary initial process definition template is used by the machine to perform an initial program load.

No check is made and no exception is signaled if the values in the template are invalid; however, the next initial program load will not be successful. *(Ref #5.)*

Hex 0108

Machine initialization status record

The MISR (machine initialization status record) is used to report the status of the machine. The status is initially collected at IPL and then updated as system status changes.

The materialize format of the MISR is as follows:

Offset		
Dec	Hex	
	Field Name	
	Data Type and Length	
8	8	
	MISR status	
	Char(6)	
8	8	
	Restart IMPL	
	Bit 0	
	0 =	IMPL was not initiated by the Terminate instruction
	1 =	IMPL was initiated by the Terminate instruction
8	8	
	Manual power on	
	Bit 1	
	0 =	Power on not due to Manual power on
	1 =	Manual power on occurred
8	8	
	Timed power on	
	Bit 2	
	0 =	Power on not due to Timed power on
	1 =	Timed power on occurred
8	8	
	Remote power on	
	Bit 3	
	0 =	Power on not due to remote power on
	1 =	Remote power on occurred
8	8	

Hex 0118

Uninterruptible power supply delay time and calculated delay time. Note: The UPS delay time is meaningful only if a UPS is installed.

The format of the template for the uninterruptible power supply delay time (including the 8-byte prefix) is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	UPS Delay time
	Bin(4)
12	C
	Calculated UPS Delay time
	Bin(4)
	The delay time interval is the amount of time the system waits for the return of utility power. If a utility power failure occurs, the system will continue operating on the UPS supplied power. If utility power does not return within the user specified delay time, the system will perform a quick power down. The delay time interval is set by the customer. The calculated delay time is determined by the amount of main storage and DASD that exists on the system. Both values are in seconds.
16	10
	—- End —-

(Ref #7.)

Hex 012C

Vital product data

The VPD (vital product data) is a template that contains information for memory card VPD, processor VPD, Columbia/Colomis VPD, central electronic complex (CEC) VPD and the panel VPD. The VPD information that is materialized is the same for all partitions of a physical machine.

The materialize format of the VPD (Including the 8-byte prefix) is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Reserved
	Char(8)
16	10
	System VPD location
	Char(32)
16	10
	Offset to memory VPD
	Bin(4)
20	14
	Offset to processor VPD
	Bin(4)
24	18
	Offset to Columbia/Colomis
	Bin(4)
28	1C
	Offset to CEC VPD
	Bin(4)
32	20
	Offset to panel VPD
	Bin(4)
36	24
	Reserved
	Char(12)
48	30
	Main store memory VPD
	Char(1040)
48	30
	Usable memory installed
	Bin(2)
	(In megabytes)
50	32
	Minimum memory required

Hex 0130

Network attributes

The network attributes is a template that contains information concerning APPN network attributes.

The materialize format of the network attributes is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Network data
	Char(190)
8	8
	System name
	Char(8)
16	10
	System name length
	Bin(2)
18	12
	New system name
	Char(8)
26	1A
	New system name length
	Bin(2)
28	1C
	Local system network identification
	Char(8)
36	24
	Local system network identification length
	Bin(2)
38	26
	End node data compression
	Bin(4)
42	2A
	Intermediate node data compression
	Bin(4)
46	2E
	Reserved
	Char(2)
48	30
	Local system control point name
	Char(8)
56	38
	Local system control point name length
	Bin(2)
58	3A
	Maximum APPN LUDs on virtual APPN CDs

Hex 0134

Date format

The date format is the format in which the date will be presented to the customer. The possible values are YMD, MDY, DMY where Y = Year, M = Month, D = Day and JUL = Julian.

The format of the template for date format is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Date format
	Char(3)
11	B
	—- End —-

(Ref #10.)

Hex 0138

Leap year adjustment

The leap year adjustment is added to the leap year calculations to determine the year in which the leap should occur. The valid values are 0, 1, 2, 3.

The format of the template for leap year adjustment is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Leap year adjustment
	Bin(2)
10	A
	—- End —-

(Ref #11.)

Hex 013C

Timed power on

The timed power on is the time and date at which the system should automatically power on if it is not already powered on. If the physical machine is powered off, and the time and date at which a partition is automatically powered is set to occur before the time and date at which the physical machine is set to automatically power on (according to their respective time of day clocks), then the partition will be powered on when the physical machine is powered on.

The format of the template for timed power on is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Minute
	Bin(2)
10	A
	Hour
	Bin(2)
12	C
	Day
	Bin(2)
14	E
	Month
	Bin(2)
16	10
	Year
	Bin(2)
18	12
	—- End —-

(Ref #12.)

Hex 0140

Timed power on enable/disable

The timed power on enable/disable allows the timed power on function to be queried to determine if the function is enabled or disabled.

The format of the template for timed power on enable/disable is as follows:

Offset

Dec Hex

Field Name

Data Type and Length

8 8

Enable/disable

Bin(2)

Hex 8000 =
Timed power on is enabled

Hex 0000 =
Timed power on is disabled

10 A

—- End —-

(Ref #13.)

Hex 0144

Remote power on enable/disable

The remote power on enable/disable allows the remote power on function to be queried to determine if the function is enabled or disabled.

The format of the template for remote power on enable/disable is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Enable/disable
	Bin(2)
	Hex 8000 = Remote power on is enabled
	Hex 0000 = Remote power on is disabled
10	A
	—- End —-

(Ref #14.)

Hex 0148

Auto power restart enable/disable

The auto power restart enable/disable allows the auto power restart function to be queried to determine if the function is enabled or disabled.

The format of the template for auto power restart enable/disable is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Enable/disable
	Bin(2)
	Hex 8000 = Auto power restart is enabled
	Hex 0000 = Auto power restart is disabled
10	A
	—- End —-

(Ref #15.)

Hex 014C

Date separator

The date separator is used when the date is presented to the customer. The valid values are a slash(/), dash(-), period(.), comma(,) and a blank( ).

The format of the template for the date separator is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Date separator
	Char(1)
9	9
	—- End —-

(Ref #16.)

Hex 0151

System security indicators (Can only be materialized)

The *system security indicators* return the current setting of the system security flags.

The format of the template for *system security indicators* is as follows:

Offset		
Dec	Hex	
	Field Name	
	Data Type and Length	
8	8	
	System security indicators	
	Char(1)	
8	8	
	Reserved	
	Bits 0-4	
8	8	
	Restrict change of service tool user ID passwords by operating system	
	Bit 5	
	0 =	Not restricted
	1 =	Restricted
8	8	
	Restrict change of operating system security values	
	Bit 6	
	0 =	Not restricted
	1 =	Restricted
8	8	
	Restrict adds to digital certificates store	
	Bit 7	
	0 =	Not restricted
	1 =	Restricted
9	9	
	Reserved (binary 0)	
	Char(11)	
20	14	
	—- End —-	

Hex 0161

Perform hardware checks on IPL

The perform hardware checks on IPL option retrieves the current setting that indicates if the system is skipping hardware checks on all physical machine IPLs. The IPL checks are performed only when the physical machine is IPLed.

The format of the template for perform hardware checks is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Perform/not perform
	Bin(2)
	Hex 8000 = The system is doing hardware checks on IPLs.
	Hex 0000 = The system is not checking the hardware on IPLs.
10	A
	—- End —-

(Ref #18.)

Hex 0164

Uninterruptible power supply type

Note: The *UPS type* is meaningful only if a UPS is installed.

The uninterruptible power supply type option allows the MI user to tell the machine how much of the system is powered by a UPS (ie, what type of UPS is installed). A *full UPS* will power all racks in the system. A *limited UPS* will have enough power to perform main store dump. A *mini UPS* will power the racks containing the CEC and the load source.

The format of the template for UPS Type is as follows:

Offset

Dec Hex

Field Name

Data Type and Length

8 8

UPS type

Bin(2)

Hex 0000 =
Indicates a full UPS is installed (all racks have a UPS installed)

Hex 4000 =
Indicates a limited UPS is installed (the UPS only has enough power to do a main store dump)

Hex 8000 =
Indicates a mini UPS is installed (only the minimum number of racks are powered)

10 A

—- End —-

(Ref #19.)

Hex 0168

Panel status request

The panel status request option returns the current status of the operations panel.

The format of the template for panel status request is as follows (including the usual 8-byte prefix):

Offset

Dec Hex

Field Name

Data Type and Length

8 8

Current IPL type

Char(1)

9 9

Panel status

Char(2)

9 9

Uninterrupted power supply installed

Bit 0

0 = UPS not installed

1 = UPS installed, ready for use

9 9

Utility power failed, running on UPS

Bit 1

0 = Running on utility power

1 = Running on UPS

9 9

Uninterrupted power supply (UPS) bypass active

Bit 2

0 = UPS bypass not active

1 = UPS bypass active

9 9

Uninterrupted power supply (UPS) battery low

Bit 3

0 = UPS battery not low

1 = UPS battery low

Hex 016C

Extended machine initialization status record

The XMISR (extended machine initialization status record) is used to report the status of the machine.

The materialize format of the XMISR is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Save storage status
	Char(4)
8	8
	Reserved (binary 0)
	Bit 0
8	8
	Completion status
	Bit 1

0 = Save storage did not complete

1 = Save storage completed

8	8
	System restored status
	Bit 2

0 = Save storage did not restore the system

1 = Save storage restored the system

8	8
	Save storage attempted
	Bit 3

0 = Save storage not attempted

1 = Save storage was attempted

8	8
	Unreadable sectors
	Bit 4

0 = Unreadable sectors were not found

1 = Unreadable sectors were found during save operation

Hex 0170

Alternate initial process definition template

The alternate initial process definition template is used by the machine when performing an automatic install.

No check is made and no exception is signaled if the values in the template are invalid; however, the next automatic install will not be successful. *(Ref #22.)*

Hex 0178

Hardware storage protection enforcement state

Note: Hardware storage protection is meaningful only on version 2 hardware or later. Hardware storage protection is not supported at all on version 1 hardware.

The hardware storage protection (HSP) mechanism is always in effect. However, HSP is enforced for individual storage areas in two different ways. For some storage areas, HSP is always enforced. For others, HSP is enforced only when this machine attribute is active. Attempted use of any storage area in a manner inconsistent with its storage protection attributes will result in *object domain or hardware storage protection violation* (hex 4401) exception when HSP is being enforced for that storage.

System objects for which HSP is always enforced are:

- programs (object type hex 02)
- modules (object type hex 03)
- XOM objects (object type hex 20)
- any objects with type values greater than hex 20.

HSP is also always enforced for secondary associated spaces. In addition, some individual objects of type space or index, and the primary associated spaces of all MI objects, can optionally be protected with HSP enforcement at all times.

The format of the template for the *hardware storage protection enforcement state* option is as follows (including the usual 8-byte prefix):

Offset

Dec Hex

Field Name

Data Type and Length

8 8

Hardware storage protection enforcement state

Bin(2)

Hex 0000 =
Indicates hardware storage protection is enforced only for storage that is always protected

Hex 8000 =
Indicates hardware storage protection is enforced for all storage

10 A

—- End —-

(Ref #23.)

Hex 0180 **Time separator**

The time separator is used when the time is presented to the customer. The valid values are a colon(:), period(.), comma(,) and a blank( ).

The format of the template for the time separator is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Time separator
	Char(1)
9	9
	—- End —-

(Ref #24.)

Hex 0184 **Software error logging**

The software error logging machine attribute is used to allow the MI user to determine whether or not software error logging is active for the machine

The format of the template for software error logging is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Software error logging
	Bin(2)
	Hex 8000 = Software error logging is active
	Hex 0000 = Software error logging is not active
10	A
	—- End —-

(Ref #25.)

Hex 0188

Machine task or secondary thread termination event control option

The machine task or secondary thread termination event option controls whether the machine will signal events when machine tasks or secondary threads terminate. The default, which is established every IPL, is to signal neither machine task nor secondary thread termination events.

There are different events associated with the termination of machine tasks and secondary threads. The machine task or secondary thread termination event option is a bit mask, with individual bits corresponding to machine tasks or secondary threads, and their associated events. If a bit is binary 1, the corresponding event will be signalled; if binary 0, it will not.

The events are signalled to the process containing the thread which most recently executed Modify Machine Attributes (MODMATR), specifying the machine task or secondary thread termination event control option attribute selection. If a process terminates while it is the process to which the machine task or secondary thread termination events are to be signalled, the signalling of these events is stopped.

The format of the template for the machine task or secondary thread termination event option is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Machine task or secondary thread termination event option
	Char(2)
8	8
	Signal machine task termination events
	Bit 0
8	8
	Signal secondary thread termination events
	Bit 1
8	8
	Reserved (binary 0)
	Bits 2-15
10	A
	—- End —-

(Ref #26.)

Hex 01A8

Service attributes

The service attributes is a template that contains system serviceability information.

The materialize format of the service attributes (including the 8-byte prefix) is as follows:

Offset		
Dec	Hex	Field Name	Data Type and Length
8	8	Service attribute flags	Char(1)
8	8	Automatic problem analysis	Bit 0

0 = Automatic problem analysis is not enabled

1 = Automatic problem analysis is enabled

8	8	Automatic problem notification	Bit 1

0 = Automatic problem notification is not enabled

1 = Automatic problem notification is enabled

8	8	Service attributes status	Bit 2

0 = Service attribute values are not set

1 = Service attribute values are set

8	8	Allow remote service access	Bit 3

0 = Remote service access is not allowed

1 = Remote service access is allowed

8	8	Allow auto service processor reporting	

Hex 01B0

Signal controls

The materialization format of the Signal Controls (including the 8-byte prefix) is as follows:

Offset

Dec	Hex	Field Name	Data Type and Length
8	8	Reserved	Char(8)
16	10	Signal blocking mask	Char(8)
16	10	Reserved (binary 0)	Bit 0
16	10	Blocked/unblocked option	Bits 1-63

0 = Signal is blocked. Signal action for the signal monitor is to be deferred.

1 = Signal is unblocked. Signal action for the signal monitor is eligible to be scheduled.

24	18	Number of signal monitors	Bin(4)
28	1C	Reserved (binary 0)	Char(4)
32	20	Signal monitor data	[*] Char(16)

(repeated for each signal monitor)

32	20	Signal number	Bin(4)
36	24	Signal action	Bin(2)

-1 = Signal associated with this signal monitor is

Hex 01C8

Cryptography attributes

The format of the template for cryptography attributes is as follows:

Offset

Dec	Hex
	Field Name
	Data Type and Length
8	8
	Number of algorithm entries to follow
	UBin(2)
10	A
	Algorithm entry
	[*] Char(6)
	(repeated *number of algorithm entries to follow* times)
10	A
	Algorithm identifier
	Char(2)

Hex 0001 =
MAC - Message Authentication Code

Hex 0002 =
MD5

Hex 0003 =
SHA-1 - Secure Hash Algorithm

Hex 0004 =
DES (encrypt only) - Data Encryption Standard

Hex 0005 =
DES (encrypt and decrypt)

Hex 0006 =
RC4

Hex 0007 =
RC5

Hex 0008 =
DESX

Hex 0009 =
Triple-DES

Hex 000A =
DSA - Digital Signature Algorithm

Hex 000B =
RSA - Rivest-Shamir-Adleman

Hex 000C =
Diffie-Hellman

Hex 000D =
CDMF - Commercial Data Masking Facility

Hex 000E =
RC2

Hex 000F =
AES - Advanced Encryption Standard

Hex 01D0

Communication network attributes (can be materialized and modified)

The communication network attribute is a template that contains information concerning communication attributes.

The format of the template for the communication network attributes is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Communication attribute
	Char(256)
8	8
	Modem country identifier
	UBin(4)
12	C
	Reserved (binary 0)
	Char(252)
264	108
	—- End —-

The **modem country identifier** specifies the country-specific identifier for modems which are internal to I/O Adapters. This value must be configured correctly to ensure proper operation and, in some countries, to meet legal requirements. There can only be one *modem country identifier* for each partition of a physical machine

The supported *modem country identifiers* are as follows:

Country	Modem Country ID (Hex value)
Argentina	00004152
Aruba	00004157
Australia	00004155
Austria	00004154
Bahrain	00004248
Belgium	00004245
Brazil	00004252
Brunei	0000424E
Canada	00004341
Cayman Islands	00004B59
Chile	0000434C
China	0000434E
Colombia	0000434F
Costa Rica	00004352

Hex 01DC

Installed processor count

This option makes available the installed processor count for the physical machine. The materialization format of installed processor count information (including the 8-byte prefix for *number of bytes provided* and *number of bytes available*) is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Number of installed processors
	UBin(2)
10	A
	—- End —-

Number of installed processors is the number of processors installed on the physical machine.

If the physical machine has the on-demand processors feature, *number of installed processors* = number of permanently activated processors + number of temporarily activated processors + number of processors which are not activated. *(Ref #31.)*

Hex 01E0

Partitioning information

This option makes available partitioning information for the physical machine and the current partition. The materialization format of partitioning information (including the 8-byte prefix for *number of bytes provided* and *number of bytes available*) is as follows:

Offset

Dec	Hex	Field Name	Data Type and Length
8	8	Current number of partitions	Char(1)
9	9	Current partition identifier	Char(1)
10	A	Primary partition identifier	Char(1)
11	B	Service partition identifier	Char(1)
12	C	Firmware level	Char(1)
13	D	Reserved (binary 0)	Char(3)
16	10	Logical serial number	Char(10)
26	1A	Reserved (binary 0)	Char(5)
31	1F	Partition attributes	Char(1)
31	1F	Partition physical processor sharing attribute	Bit 0

0 = Partition does not share physical processors

1 = Partition shares physical processors

31	1F	Partition uncapped attribute	

Hex 01EC

Additional load source reserved space (Can be modified and materialized)

Use this selection to check if additional load source disk space is allowed to be reserved for system use and if that space has already been reserved. If the indicators show that additional load source disk space is allowed for system use and that LIC has not reserved the space yet, after IPL, LIC will reserve this space and it cannot be freed.

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Reserved disk space indicators
	Char(1)
8	8
	Space reserved indicator
	Bit 0
	This indicator indicates what can take effect on the next IPL.
	0 = LIC cannot reserve load source disk space.
	1 = LIC can reserve load source disk space on the next IPL.
8	8
	Load source space reserved
	Bit 1
	0 = LIC has not reserved load source disk space.
	1 = LIC has reserved load source disk space.
8	8
	Reserved (binary 0)
	Bits 2-7
9	9
	—- End —-

The **space reserved indicator** field indicates whether additional load source disk space can be reserved by LIC or not on the next IPL. Once the space is reserved by LIC, that is, the value of *load source space reserved* is binary 1, the value of this field no longer has any meaning.

The **load source space reserved** field indicates whether additional load source disk space has been reserved by LIC or not. *(Ref #33.)*

Hex 01F4

On-demand processor information (Can only be materialized)

Use this selection to materialize the information of on-demand processors on the system.

If the system does not have the on-demand processor feature installed, all non-reserved values returned will be blanks (hex 40s) except the enabled and active indicators will be hex 00s, and the *current time of day* will be set. Also, some features may support limited on-demand functions, such as *Capacity Upgrade on Demand* (CUoD) but not *On/Off Capacity on Demand* (CoD) for processors. In such a case, the values returned for the unsupported function will be blanks (hex 40s).

The format of the template is as follows:

Offset

Dec	Hex	Field Name	Data Type and Length
8	8	System type	Char(4)
12	C	System serial number	Char(10)
22	16	Capacity card CCIN	Char(4)
26	1A	Capacity card serial number	Char(10)
36	24	Capacity card unique identifier	Char(16)
52	34	Capacity Upgrade on Demand activation feature	Char(4)
56	38	Activated Capacity Upgrade on Demand units	Char(4)
60	3C	Capacity Upgrade on Demand sequence number	Char(4)
64	40	Capacity Upgrade on Demand entry check	Char(2)
66	42	Capacity Upgrade on Demand maximum processors that can be purchased	Char(4)
70	46	On/Off Capacity on Demand enabled	

Hex 01F6

On-demand memory information (Can only be materialized)

Use this selection to materialize the information of on-demand memory on the system.

If the system does not have the on-demand memory feature installed, all non-reserved values returned will be blanks (hex 40s) except the enabled and active indicators will be hex 00s, and the *current time of day* will be set. Also, some features may support limited on-demand functions, such as Capacity Upgrade on Demand but not On/Off Capacity on Demand for memory. In such a case, the values returned for the unsupported function will be blanks (hex 40s).

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	System type
	Char(4)
12	C
	System serial number
	Char(10)
22	16
	Capacity card CCIN
	Char(4)
26	1A
	Capacity card serial number
	Char(10)
36	24
	Capacity card unique identifier
	Char(16)
52	34
	Capacity Upgrade on Demand activation feature
	Char(4)
56	38
	Activated Capacity Upgrade on Demand units
	Char(4)
60	3C
	Capacity Upgrade on Demand sequence number
	Char(4)
64	40
	Capacity Upgrade on Demand entry check
	Char(2)
66	42
	Capacity Upgrade on Demand maximum memory that can be purchased
	Char(4)
70	46
	On/Off Capacity on Demand enabled

Hex 01F7

On-demand memory information (Can only be materialized) (Internal use only)

This option can only be used in a program running in system state. The *scalar value invalid* (hex 3203) exception will be signaled if the program that issues this option is running in user state.

Use this selection to materialize the information of on-demand memory on the system, including information required to verify resource usage.

If the system does not have the on-demand memory feature installed, all non-reserved values returned will be blanks (hex 40s) except the enabled and active indicators will be hex 00s, and the *current time of day* will be set. Also, some features may support limited on-demand functions, such as Capacity Upgrade on Demand but not On/Off Capacity on Demand for memory. In such a case, the values returned for the unsupported function will be blanks (hex 40s).

The format of the template is as follows:

Offset

Dec	Hex	Field Name	Data Type and Length
8	8	System type	Char(4)
12	C	System serial number	Char(10)
22	16	Capacity card CCIN	Char(4)
26	1A	Capacity card serial number	Char(10)
36	24	Capacity card unique identifier	Char(16)
52	34	Capacity Upgrade on Demand activation feature	Char(4)
56	38	Activated Capacity Upgrade on Demand units	Char(4)
60	3C	Capacity Upgrade on Demand sequence number	Char(4)
64	40	Capacity Upgrade on Demand entry check	Char(2)
66	42		

Hex 01F8

IPL identifier (Can only be materialized)

Use this selection to materialize the IPL identifier. This value changes with each IPL.

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	IPL identifier
	UBin(4)
12	C
	—- End —-

The **IPL identifier** is a value that is unique for each system IPL. The value increases for each system IPL. *(Ref #37.)*

Hex 01FC

Electronic licensing identifier (Can be modified and materialized)

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Electronic licensing identifier
	Char(5)
13	D
	—- End —-

The **electronic licensing identifier** field is the value of version, release and modification level of the OS/400 to be installed during the next upgrade whose license is accepted by the customers. The format of the *electronic licensing identifier* is vrmnn where v is the version, r, the release, m, the modication level, and nn are operating system assigned values. *(Ref #38.)*

Hex 0200

Wait state performance information (Can only be materialized)

Use this selection to materialize the wait state performance information.

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Reserved (binary 0)
	Char(8)
16	10
	Offset to descriptor table entries
	UBin(4)
20	14
	Offset to mapping table entries
	UBin(4)
24	18
	Number of descriptor table entries
	UBin(2)
26	1A
	Number of mapping table entries
	UBin(2)
28	1C
	Reserved (binary 0)
	Char(20)
48	30
	Descriptor table entry
	[*] Char(64)
	(repeated *number of descriptor table entries* times)
48	30
	Collection bucket number
	UBin(2)
50	32
	Collection bucket descriptor
	Char(50)
100	64
	Reserved (binary 0)
	Char(12)
*	*
	Mapping table entry
	[*] Char(16)
	(repeated *number of mapping table entries* times)

Hex 0204

Hardware management console information

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Number of entries returned
	UBin(4)
12	C
	Reserved (binary 0)
	Char(4)
16	10
	Hardware Management Console (HMC) information
	[*] Char(1036)
	(repeated for *number of entries returned*)
16	10
	HMC information length
	UBin(2)
18	12
	HMC information
	Char(1034)
*	*
	—- End —-

The **number of entries returned** field returns the number of *Hardware Management Console (HMC) information* entries returned. On a non-HMC managed system, the value returned will be binary 0.

The **HMC information** field returns a string containing the following data:

- HMC name
- HMC host name
- IP address
- HMC state

The data returned is in 7-bit ASCII and its format is as follows: keyword1=its_value;keyword2=its_value;etc. where a keyword can be HscName, HscHostName, HscIPAddr, or HmcStat. Each keyword is followed by an equal sign (=), its value, and ends with a semi-colon (;). For example, an HMC information string can look like this: HscName=679231U*23WW193; HscHostName=hosta.company.xyz.com; HscIPAddr=3.103.123.118;HmcStat=1;

Keywords and their values in the *HMC information* string can be in any order. There is no carriage return <CR> (hex 0D) or line feed <LF> (hex 0A), and the string is not NULL terminated.

The values of HMC state can be

- 1 = the HMC is operating successfully.
- 2 = the HMC has indicated it is temporarily disconnecting.

Hex 0208

Keep current disk configuration during install (can be modified and materialized)

This option is used to materialize the indicator that indicates whether or not all non configured disk units should be added to the system ASP during an automatic install operation.

The format of the template is as follows:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
8	8
	Keep current disk configuration indicator
	Char(1)
	Hex 00 = Add all non configured disk units to the system ASP.
	Hex 01 = Keep current disk configuration.
9	9
	Reserved (binary 0)
	Char(7)
16	10
	—- End —-

Limitations (Subject to Change): Data-pointer-defined scalars are not allowed as a primary operand for this instruction. An *invalid operand type* (hex 2A06) exception is signaled if this occurs.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C0A Service Processor Unable to Process Request

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Machine Data (MATMDATA)

Op Code (Hex)	Operand 1	Operand 2
0522	Receiver	Materialization option

Operand 1: Character variable scalar.

Operand 2: Character(2) constant, or unsigned binary(2) constant or immediate.

Bound program access

```
Built-in number for MATMDATA is 160.
MATMDATA (
          receiver                : address
          materialization_option  : literal(2) OR
                                    literal(4)
)
```

The *materialization_option* may be declared as a literal of any scalar data type.

```
  -- OR --

Built-in number for MATTOD is 94.

MATTOD (
        time_of_day   : address
)
```

The time-of-day clock is materialized. This function is identical to MATMDATA when a *materialization_option* value of Hex 0000 is specified.

Description: The machine data requested by *materialization option* is returned at the location specified by *receiver*. For the purposes of this instruction, machine data refers to any data that is encapsulated by the machine. The data can be either thread-specific or apply system-wide.

Operand 2 is a 2-byte value. The value of operand 2 determines which machine data are materialized. Operand 2 is restricted to a constant character or unsigned binary scalar or an immediate value. A summary of the allowable values for Operand 2 follows.

Table 1. Materialization option

Option value	Description	Page
Hex 0000	Materialize time-of-day clock as local time	″Hex 0000 = Materialize time-of-day clock as local time″ (page)
Hex 0001	Materialize system parameter integrity validation flag	″Hex 0001 = Materialize system parameter integrity validation flag″ (page)
Hex 0002	Materialize thread execution mode flag	″Hex 0002 = Materialize thread execution mode flag″ (page)

Option value	Description	Page
Hex 0003	Materialize maximum size of a space object or associated space when space alignment is chosen by the machine	″Hex 0003 = Materialize maximum size of a space object or associated space when space alignment is chosen by the machine″ (page)
Hex 0004	Materialize time-of-day clock as Coordinated Universal Time (UTC)	″Hex 0004 = Materialize time-of-day clock as Coordinated Universal Time (UTC)″ (page)
Hex 0005 though FFFF	Reserved	

Operand 1 specifies a *receiver* into which the materialized data is placed. It must specify a character scalar with a minimum length which is dependent upon the *materialization option* specified for operand 2. The *receiver* may be substringed. The start position of the substring may be a variable. However, the length of the substring must be an immediate or constant. The length specified for operand 1 must be at least the required minimum. Only the bytes up to the required minimum length are used. Any excess bytes are ignored.

The data placed into the *receiver* differs depending upon the *materialization option* specified. The following descriptions detail the formats of the optional materializations.

Hex 0000 = Materialize time-of-day clock as local time: (minimum *receiver* length is 8)

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Time of day	Char(8)
8	8	—- End —-	

Time of day is the time value of the time-of-day clock which is returned as the local time for the system. See “Standard Time Format” on page 1272 for a detailed description of the format for a time value.

Unpredictable results occur if the time-of-day clock is materialized before it is set.

The time-of-day clock can be materialized as the Coordinated Universal Time (UTC) for the system using ″Hex 0004 = Materialize time-of-day clock as Coordinated Universal Time (UTC)″ (page).

See “Time-of-Day (TOD) Clock” on page 1273 for detailed descriptions of the time-of-day clock, local time, and UTC.

Performance note: The time-of-day clock may be materialized, with the *time of day* returned as the local time for the system, with this instruction and also with the Materialize Machine Attributes (MATMATR) instruction. Better performance may be realized with the use of this instruction rather than with the MATMATR instruction.

Hex 0001 = Materialize system parameter integrity validation flag: (minimum *receiver* length is 1)

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	System parameter integrity validation flag	Char(1)
1	1	—- End —-	

This option returns the value of the machine attribute which specifies whether additional validation of parameters passed to programs which run when the thread is in system state is to be performed, such as for U. S. government's Department of Defense security ratings.

A value of hex 01 indicates this additional checking is being performed. A value of hex 00 is returned otherwise.

Hex 0002 = Materialize thread execution mode flag: (minimum *receiver* length is 1)

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Thread execution mode flag	Char(1)
1	1	—- End —-	

This option returns the value of the thread execution mode for the thread in which the instruction is run.

A returned value of hex 01 indicates that thread is currently executing in kernel mode. A value of hex 00 is returned otherwise.

Hex 0003 = Materialize maximum size of a space object or associated space when space alignment is chosen by the machine: (minimum *receiver* length is 4)

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Maximum size of machine-aligned space object or associated space	UBin(4)
4	4	—- End —-	

This option returns the maximum size in bytes of a space object or associated space created with the space alignment chosen by the machine. Some types of objects may not support an associated space of the maximum size.

This size may vary with each machine implementation.

Hex 0004 = Materialize time-of-day clock as Coordinated Universal Time (UTC): (minimum *receiver* length is 8)

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Time of day	Char(8)
8	8	—- End —-	

Time of day is the time value of the time-of-day clock which is returned as the Coordinated Universal Time (UTC) for the system. See "Standard Time Format" on page 1272 for a detailed description of the format for a time value.

Unpredictable results occur if the time-of-day clock is materialized before it is set.

The time-of-day clock can be materialized as the local time for the system using "Hex 0000 = Materialize time-of-day clock as local time" (page 694).

See "Time-of-Day (TOD) Clock" on page 1273 for detailed descriptions of the time-of-day clock, local time, and UTC.

Performance note: The time-of-day clock may be materialized, with the *time of day* returned as the UTC for the system, with this instruction and also with the Materialize Time Of Day Attributes (MATTODAT) instruction. Better performance may be realized with the use of this instruction rather than with the MATTODAT instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Machine Information (MATMIF)

Bound program access

`Built-in number for MATMIF is 670.`
`MATMIF (`
`  receiver  : address`
`  materialization_option  : unsigned binary(2)`
`) :  signed binary(4)  /* result */`

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release. The data returned by this instruction is subject to change each and every release. Fields may be added, deleted, reordered, and/or have their unit of measure changed. There will be no attempt made to maintain any compatibility from release to release.

Description: Information specified by the *materialization option* operand is materialized into the template addressed by the *receiver* operand. Upon successful completion, *result* is set to binary 0.

In the case where the requested data cannot be returned, the EUNKNOWN error number is returned in the *result*.

The *receiver* specifies a space that is to receive the materialized information. The space pointer specified must address a 16-byte aligned area. If not, the EFAULT error number is returned in the *result*.

The *materialization option* specifies which information is to be materialized. If an invalid value is specified, the EINVAL error number is returned in the *result*.

A summary of the allowable hex values for *materialization option* follows.

Table 1. Materialize Machine Information options

Materialization Option	Page
Hex 0001 - Option 0001 information	reference #1 (page 698)
Hex 0002 - Option 0002 information	reference #2 (page 700)

The *receiver* template has the following format

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided	UBin(4)
4	4		Number of bytes available	UBin(4)
8	8	Information	Char(*)	
*	*	—- End —-		

The first 4 bytes identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the ENOSPC error number to be returned in the *result*.

The second 4 bytes identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No error numbers (other than the ENOSPC described previously) are returned if the receiver contains insufficient area for the materialization.

The following information requires *receivers* of varying lengths. The information that will be materialized and their *materialization option* values follow. If the system does not support a function, hex zeros will be returned in that field.

•

• Hex 0001 = Option 0001 information

Offset Dec	Offset Hex	Field Name	Data Type and Length	
8	8	Maximum memory	UBin(8)	
16	10	Minimum memory	UBin(8)	
24	18	Dispatch wheel rotation period	UBin(8)	
32	20	Partition ID	UBin(4)	
36	24	Indicators	Char(4)	
36	24		Reserved (binary 0)	Bits 0-29
36	24		Bound hardware threads indicator	Bit 30
			0 = Hardware threads are not bound	
			1 = Hardware threads are bound	
36	24		Dedicated processors indicator	Bit 31
			0 = Partition shares processors	
			1 = Partition has dedicated processors	
40	28	Maximum processors in the platform	UBin(4)	
44	2C	Minimum virtual processors	UBin(4)	
48	30	Maximum virtual processors	UBin(4)	

Offset Dec	Offset Hex	Field Name	Data Type and Length
52	34	Minimum processor capacity	UBin(4)
56	38	Maximum processor capacity	UBin(4)
60	3C	Processor capacity delta	UBin(4)
64	40	Minimum interactive capacity percentage	UBin(4)
68	44	Maximum interactive capacity percentage	UBin(4)
72	48	Hardware threads per processor	UBin(2)
74	4A	Partition name	Char(256)
330	14A	Reserved (binary 0)	Char(6)
336	150	Memory delta	UBin(8)
344	158	Reserved (binary 0)	Char(8)
352	160	—- End —-	

Maximum memory is the maximum amount of memory (in units of megabytes) that can be assigned to this partition.

Minimum memory is the minimum amount of memory (in units of megabytes) that is needed in this partition.

Dispatch wheel rotation period is the number of nanoseconds in the hypervisor's scheduling window. Each virtual processor will be given the opportunity to execute on a physical processor sometime during this period. The amount of time each virtual processor is able to use on a physical processor corresponds to *processor capacity*.

Partition ID is the identifier of this partition. It is unique within a physical machine.

Bound hardware threads indicator indicates whether or not hardware threads are bound. *Hardware threads are not bound* indicates that the system can not assume that a set of hardware threads will always be dispatched together on a physical processor. *Hardware threads are bound* indicates that a set of hardware threads will always be dispatched together on a physical processor. This allows an operating system to make scheduling decisions based on cache affinity and work load.

Dedicated processors indicator indicates whether or not the partition uses only dedicated physical processors. *Partition has dedicated processors* indicates that this partition uses only dedicated physical processors. This means that each virtual processor in the partition has a corresponding entire physical processor. *Partition shares processors* indicates that this partition uses physical processors from a shared pool of physical processors. The number of virtual processors represents the maximum number of concurrent units of execution that can be active in the partition at any point in time. Each virtual processor has the processing capacity of some fraction of a physical processor. One or more partitions may be executing on the physical processors in the shared processor pool at any given point in time.

Maximum processors in the platform is the maximum number of physical processors that can be active in this platform without physically installing additional processors. This field includes currently active processors and any standby processors that are present in the platform.

Minimum virtual processors is the minimum number of virtual processors that are needed in this partition.

Maximum virtual processors is the maximum number of virtual processors that can be assigned to this partition.

Minimum processor capacity is the minimum amount of processor capacity (in units of 1/100 of a physical processor) that is needed in this partition.

Maximum processor capacity is the maximum amount of processor capacity (in units of 1/100 of a physical processor) that can be assigned to this partition.

Processor capacity delta is the delta (in units of 1/100 of a physical processor) that can be added to or removed from this partition's *processor capacity*.

Minimum interactive capacity percentage is the minimum value that can be set for this partition's *interactive capacity percentage*.

Maximum interactive capacity percentage is the maximum value that can be set for this partition's *interactive capacity percentage*.

Hardware threads per processor is the number of hardware threads per processor when *hardware multi-threading is enabled*.

Partition name is the name that has been assigned to this partition. This field is a null-terminated ASCII character string.

Memory delta is the delta (in units of megabytes) that can be added to or removed from this partition's *memory*.

- Hex 0002 = Option 0002 information

Offset Dec	Offset Hex	Field Name	Data Type and Length	
8	8	Memory	UBin(8)	
16	10	Dispatch time since IPL	UBin(8)	
24	18	Interactive time since IPL	UBin(8)	
32	20	Excess interactive time since IPL	UBin(8)	
40	28	Shared processor pool idle time since IPL	UBin(8)	
48	30	Indicators	Char(4)	
48	30		Reserved (binary 0)	Bits 0-28
48	30		Capped partition indicator	Bit 29
			0 = Uncapped partition	
			1 = Capped partition	
48	30		Hardware multi-threading indicator	Bit 30
			0 = Hardware multi-threading is not enabled	
			1 = Hardware multi-threading is enabled	
48	30		Shared processor pool idle time since IPL indicator	Bit 31
			0 = Shared processor pool idle time since IPL was not materialized	
			1 = Shared processor pool idle time since IPL was materialized	
52	34	Processors in the platform	UBin(4)	
56	38	Virtual processors	UBin(4)	
60	3C	Physical processors in the shared processor pool	UBin(4)	
64	40	Unallocated processor capacity in the shared processor pool	UBin(4)	
68	44	Processor capacity	UBin(4)	
72	48	Variable processor capacity weight	UBin(4)	
76	4C	Unallocated variable processor capacity weight	UBin(4)	
80	50	Minimum required processor capacity	UBin(4)	
84	54	Interactive capacity percentage	UBin(4)	
88	58	Partition group ID	UBin(2)	
90	5A	Shared processor pool ID	UBin(2)	
92	5C	Interactive threshold	UBin(2)	
94	5E	Reserved (binary 0)	Char(2)	
96	60	—- End —-		

Memory is the amount of memory (in units of megabytes) currently allocated to this partition.

Dispatch time since IPL is the number of nanoseconds of processor time used since IPL.

Interactive time since IPL is the number of nanoseconds of processor time used by interactive processes since IPL. An interactive process is any process doing 5250 display device I/O. For additional information on interactive processes, see manual SC41-0607 iSeries Performance Capabilities Reference manual which is available in the iSeries Information Center.

Excess interactive time since IPL is the number of nanoseconds of processor time used by interactive processes since IPL, that exceeded *interactive capacity*.

Shared processor pool idle time since IPL is the number of nanoseconds of processor time that the shared processor pool has been idle since IPL, when *shared processor pool idle time since IPL was materialized*.

Capped partition indicator indicates whether or not the partition is allowed to use more than its *processor capacity*. An *uncapped partition* is allowed to use more than its *processor capacity* if processor capacity is available in the shared processor pool. A *capped partition* is not allowed to use more than its *processor capacity* even if the partition has work to do and there is processor capacity available in the shared processor pool.

Hardware multi-threading indicator indicates whether or not hardware multi-threading is enabled. *Hardware multi-threading is enabled* indicates that hardware multi-threading is active. *Hardware multi-threading is not enabled* indicates that hardware multi-threading is inactive. Hardware multi-threading refers to the ability of a processor to be multi-threaded. When a processor is multi-threaded, multiple tasks of execution can be loaded into the same processor. Each task of execution is referred to as a hardware thread. When *hardware multi-threading is enabled*, *hardware threads per processor* indicates the number of hardware threads each processor has.

Shared processor pool idle time since IPL indicator indicates whether or not the *shared processor pool idle time since IPL* was successfully materialized. *Shared processor pool idle time since IPL was materialized* indicates that the *shared processor pool idle time since IPL* was successfully materialized and contains valid information. *Shared processor pool idle time since IPL was not materialized* indicates that the *shared processor pool idle time since IPL* was not materialized (the partition is not authorized to retrieve this information) and should not be used.

Processors in the platform is the number of physical processors in this platform that are available for customer use. This does not include temporary processors on demand that have not been turned on.

Virtual processors is the number of virtual processors in this partition.

Physical processors in the shared processor pool is the number of physical processors that are allocated to the shared processor pool in which this partition is executing.

Unallocated processor capacity in the shared processor pool is the amount of processor capacity (in units of 1/100 of a physical processor) in this partition's shared processor pool, that is available to be allocated to *processor capacity*.

Processor capacity is the amount of processor capacity (in units of 1/100 of a physical processor) currently available to the partition. For a partition using dedicated processors this value represents the number of virtual processors currently active in the partition. For a partition using shared processors this value represents this partition's share of processors from its shared processor pool.

Variable processor capacity weight is the weighting factor that is used to assign additional unused processor capacity (from the shared processor pool) to *processor capacity*. This factor will be in the range of 0 - 255. A value of 0 effectively caps this partition at its *processor capacity*.

Unallocated variable processor capacity weight is the amount of capacity weight that is available for allocation to the *variable processor capacity weight*.

Minimum required processor capacity is the amount of processor capacity (in units of 1/100 of a physical processor) that the operating system requires in this partition.

Interactive capacity percentage is this partition's portion (in hundredths of a percent) of the platform's interactive capacity. For instance if the platform was allowed to do 2000 units of interactive work per second, and this field was 5000 (50%), then this partition would be allowed to perform 1000 units of interactive work per second.

Partition group ID identifies the LPAR group that this partition is a member of. A LPAR group (aka partition group) is a set of partitions on a platform that a Work Load Manager (WLM) will manage to achieve its goals. WLM manages the partitions by moving resources from one partition in the group to another partition in the group.

Shared processor pool ID identifies the shared processor pool this partition is a member of. This field should only be used when *partition shares processors*. A shared processor pool is a set of physical processors on the platform that is used to run a set of shared processor partitions that exist on this platform.

Interactive threshold is the maximum interactive processor utilization (in hundredths of a percent) which can be sustained in this partition, without causing a disproportionate increase in system overhead. For example, a value of 2379 means that the threshold is 23.79%. On a machine with no limit on interactive utilization, the value returned will be 10000 (100%).

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Error conditions

The *result* will be set to one of the following:

EFAULT 3408 - The address used for an argument was not correct.

EINVAL 3021 - The value specified for the argument is not correct.

ENOSPC 3404 - No space available.

EUNKNOWN 3474 - Unknown system state.

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1005 Authority Verification Terminated Due to Damaged Object
- 1044 Partial System Object Damage

1A Lock State

- 1A01 Invalid Lock State

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 2203 Object Suspended
- 2207 Authority Verification Terminated Due to Destroyed Object
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

36 Space Management

- 3601 Space Extension/Truncation

44 Protection Violation

- 4401 Object Domain or Hardware Storage Protection Violation
- 4402 Literal Values Cannot Be Changed

Materialize Mutex (MATMTX)

Bound program access

```
Built-in number for MATMTX is 163.
MATMTX (
        operand1   : address
        operand2   : address
        operand3   : address of unsigned binary(4) value OR
                     null pointer value
)
```

Description: The current state of the mutex or replica of a mutex whose address is passed in operand 2 is materialized into the receiver space identified by operand 1. A replica of a mutex can be returned by the MATPRMTX instruction. The space pointed to by operand 3 is a 4-byte unsigned binary field used to indicate the type of information that should be returned by this instruction.

The mutex must be aligned on a 16-byte boundary.

The **materialization options** value referenced by operand 3 has the following format:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Reserved (binary 0)	Bits 0-29
0	0	Mutex attributes option	Bit 30
		0 = Do not return additional mutex attributes	
		1 = Return additional mutex attributes	
0	0	Reserved (binary 0)	Bit 31
4	4	—- End —-	

The **mutex attributes option** field is used to select whether or not additional mutex attributes are to be returned. If *mutex attributes option* is set to *do not return additional mutex attributes*, then a standard materialization template is used. If *mutex attributes option* is set to *return additional mutex attributes*, then a materialization template with extended *mutex descriptors* is used.

If operand 3 contains a null pointer value, the default *materialization options* are used. All values other than those specifically defined for *materialization options* are reserved and will cause a *scalar value invalid* (hex 3203) exception to be generated.

The materialization template identified by operand 1 must be 16-byte aligned. If the materialization template is not properly aligned, a *boundary alignment* (hex 0602) exception is signaled. The format of the information returned in the materialization template is different, depending on the *materialization options* selected.

The materialization template has the following standard format when *mutex attributes option* is set to *do not return additional mutex attributes*:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Reserved (binary 0)	Char(4)	
12	C	Number of waiters	Bin(4)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Mutex name	Char(16)	
32	20	Mutex owner	Char(30)	
62	3E	Reserved (binary 0)	Char(18)	
80	50	Wait list descriptors (repeated for each thread waiting for mutex)	[*] Char(48)	
80	50		Process identifier	Char(30)
110	6E		Reserved (binary 0)	Char(18)
*	*	—- End —-		

The first 4 bytes of the materialization identifies the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identifies the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver that can be used to completely fill *wait list descriptors*. Partial descriptors are not returned. If the *number of bytes provided* would cause the storage boundary of the space provided for the receiver to be exceeded, and if the *number of bytes available* would actually exceed this boundary, then a *space addressing violation* (hex 0601) exception is signaled. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception described previously.

The **number of waiters** is the number of threads that are currently waiting for the mutex to become unlocked.

Mutex name contains the name of the mutex. The name is left-justified and padded to the right with blanks. If the mutex was created using a null-terminated name string, the name materialized with this instruction is null-terminated instead of padded with blanks. If the mutex was created without a name, this field will contain the character string "UNNAMED_" followed by the first 8 characters of the program which created the mutex.

The **mutex owner** contains the name of the process containing the thread that holds the mutex lock. If this field is all blanks, the mutex is not locked. The name returned here is the 30-character process control space name.

The **wait list descriptors** identify the threads waiting for the mutex to become unlocked. **Process identifier** contains the name of the process containing the waiting thread. The name returned here is the 30-character process control space name.

The materialization template has the following extended format when *mutex attributes option* is set to *return additional mutex attributes*:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Reserved (binary 0)	Char(4)	
12	C	Number of waiters	Bin(4)	
16	10	Mutex name	Char(16)	
32	20	Mutex owner	Char(30)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
62	3E	Reserved (binary 0)	Char(2)	
64	40	Mutex owner thread ID	Char(8)	
72	48	Mutex owner unique thread value	Char(8)	
80	50	Wait list descriptors (repeated for each thread waiting for mutex)	[*] Char(48)	
80	50		Process identifier	Char(30)
110	6E		Reserved (binary 0)	Char(2)
112	70		Waiter thread ID	Char(8)
120	78		Waiter unique thread value	Char(8)
*	*	—- End —-		

The contents of the template fields are as defined for the previous template(s), unless specifically defined or redefined as follows:

The **mutex owner** contains the name of the process containing the thread that holds the mutex lock. If this field is all blanks, the mutex is not locked. The name returned here is the 30-character process control space name. The **mutex owner thread ID** contains a process specific value that identifies the thread within the process that holds the mutex lock. If this field is binary 0, the mutex is not locked. The **mutex owner unique thread value** contains a system-wide unique value that identifies the specific thread that holds the mutex lock. If this field is binary 0, the mutex is not locked. This field cannot be used as input on any other MI instruction, but may be useful for debug purposes.

The **wait list descriptors** identify the threads waiting for the mutex to become unlocked. **Process identifier** contains the name of the process containing the waiting thread. The name returned here is the 30-character process control space name. The **waiter thread ID** contains a process specific value that identifies the thread within the process that is waiting for the mutex. The **waiter unique thread value** contains a system-wide unique value that identifies the specific thread that is waiting for the mutex. This field cannot be used as input on any other MI instruction, but may be useful for debug purposes.

An *invalid mutex* (hex 3804) exception is generated if an attempt is made to materialize a mutex that does not exist.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

3804 Invalid Mutex

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Object Locks (MATOBJLK)

Op Code (Hex)	Operand 1	Operand 2
033A	Receiver	System object or space location

Operand 1: Space pointer.

Operand 2: System pointer or space pointer data object.

Bound program access

```
Built-in number for MATOBJLK is 50.
MATOBJLK (
          receiver                          : address
          system_object_or_space_location   : address of system pointer OR
                                              address of space pointer(16)
)
```

Description: If operand 2 is a system pointer, the current lock status of the object identified by the system pointer is materialized into the template specified by operand 1. If operand 2 is a space pointer, the current lock status of the specified space location is materialized into the template specified by operand 1. The materialization template identified by operand 1 must be aligned on a 16-byte boundary. The format of the materialization is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Current cumulative lock status	Char(3)	
8	8		Lock states currently allocated (1 = yes)	Char(1)
8	8		LSRD	Bit 0
8	8		LSRO	Bit 1
8	8		LSUP	Bit 2
8	8		LEAR	Bit 3
8	8		LENR	Bit 4
8	8		Locks implicitly set	Bit 5
8	8		Reserved (binary 0)	Bits 6-7
9	9		Lock states for which threads are in synchronous wait (1 = yes)	Char(1)
9	9		LSRD	Bit 0
9	9		LSRO	Bit 1
9	9		LSUP	Bit 2
9	9		LEAR	Bit 3

Offset Dec	Offset Hex	Field Name	Data Type and Length	
9	9		LENR	Bit 4
9	9		Implicit lock request	Bit 5
9	9		Reserved (binary 0)	Bits 6-7
10	A		Lock states for which threads are in asynchronous wait (1 = yes)	Char(1)
10	A		LSRD	Bit 0
10	A		LSRO	Bit 1
10	A		LSUP	Bit 2
10	A		LEAR	Bit 3
10	A		LENR	Bit 4
10	A		Reserved (binary 0)	Bits 5-7
11	B	Reserved (binary 0)	Char(1)	
12	C	Number of lock state descriptions	Bin(2)	
14	E	Reserved (binary 0)	Char(2)	
16	10	Lock state descriptions (repeated *number of lock state descriptions* times)	[*] Char(32)	
16	10		Lock holder or waiter	System pointer
32	20		Lock state	Char(1)
32	20		LSRD	Bit 0
32	20		LSRO	Bit 1
32	20		LSUP	Bit 2
32	20		LEAR	Bit 3
32	20		LENR	Bit 4
32	20		Reserved (binary 0)	Bits 5-7
33	21		Status of lock request	Char(1)
33	21		Lock scope object type	Bit 0
			0 = Process control space	
			1 = Transaction control structure	
33	21		Lock scope	Bit 1
			0 = Lock is scoped to the *lock scope object type*	
			1 = Lock is scoped to the thread	
33	21		Reserved	Bit 2
33	21		Waiting because this lock is not available	Bit 3
33	21		Thread in asynchronous wait for lock	Bit 4
33	21		Thread in synchronous wait for lock	Bit 5
33	21		Implicit lock (machine applied)	Bit 6
33	21		Lock held by a process, thread or transaction control structure	Bit 7
34	22		Lock information	Char(1)
34	22		Reserved (binary 0)	Bits 0-5
34	22		Lock is held by a process, thread, or transaction control structure other than the current process or thread, or lock is waited on by some other thread	Bit 6

Offset Dec	Hex	Field Name	Data Type and Length	
34	22		Lock is held by the machine	Bit 7
35	23		Reserved (binary 0)	Char(1)
36	24		Unopened thread handle	UBin(4)
40	28		Thread ID	Char(8)
*	*	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This total is supplied as input to the instruction and is not modified by the instruction. A total of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception described previously) are signaled if the *receiver* contains insufficient area for the materialization.

Locks may be implicitly applied by the machine (**implicit lock** is binary 1). If the implicit lock is held for a process or thread, a pointer to the associated process control space is returned in the **lock holder or waiter** field. If the implicit lock is held for a transaction control structure, a pointer to the associated transaction control structure is returned in the *lock holder or waiter* field. Locks held by the machine, but not related to a specific process, thread, or transaction control structure, cause the *lock holder or waiter***unopened thread handle**, and **thread ID** fields to each be assigned a value of binary 0.

When a lock is held by a process or a thread, the system security level is 40 or greater, and the invoker of this instruction is a user state program, then the *process control space* system pointer associated with the lock will be returned in the *lock holder or waiter* field if the lock is held by the current thread or its containing process. This field will be set to binary 0 if the lock is held by some other process or thread, or if the lock is waited on by some other thread. When system security level 30 or less is in effect or when the invoking program is in system state, then the *lock requestor* field will always be returned with the appropriate *process control space* system pointer value (which may be binary 0 if the machine holds the lock).

When the invoker of this instruction is a user state program, then the *unopened thread handle* and *thread ID* fields will be returned if the lock is held or waited on by the current thread. These fields will be set to binary 0 if the lock is held by some other process, thread, or transaction control structure, or if the lock is waited on by some other thread. When the invoking program is in system state, then the *unopened thread handle* and *thread ID* fields will always be returned with the appropriate values (which may be binary 0 if the machine holds the lock or if a transaction control structure holds the lock).

Locks may be held by a transaction control structure. If **lock scope object type** has a value of *transaction control structure*, then the *lock holder or waiter* field will contain a system pointer to the transaction control structure that holds the lock and the *unopened thread handle* and *thread ID* fields will be assigned a value of binary 0. When a thread is waiting for a transaction control structure scope lock, the *lock holder or waiter*, *unopened thread handle*, and *thread ID* will identify the thread that is waiting for the lock.

The *lock information* will be set appropriately regardless of security level and program state.

Only a single *lock state* is returned for each *lock state description*.

A space pointer machine object cannot be specified for operand 2.

A *lock state description* for a lock held by a process or a transaction control structure will have a value of binary 0 for the *unopened thread handle* and for the *thread id*. A *lock state description* for a lock held by a thread will have the *lock holder or waiter* contain a system pointer to the process control space containing the thread, and a non-zero value for the *unopened thread handle* and for the *thread ID* to identify the specific thread within the process that is holding the lock. A *lock state description* for a lock being waited on will have the *lock holder or waiter* contain a system pointer to the process control space containing the waiting thread, and a non-zero value for the *unopened thread handle* and for the *thread ID* to identify the specific thread that is waiting for the lock.

The maximum number of locks that can be materialized with this instruction is 32,767. No exception will be signaled if more than 32,767 exist and only the first 32,767 locks found will be materialized.

Warning: Temporary Level 3 Header

Authorization Required

•

- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize or Verify Licensed Internal Code Options (MVLICOPT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0223	Licensed Internal Code options	Control options	Result template

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Space pointer.

Bound program access

```
Built-in number for MVLICOPT is 613.
MVLICOPT (
          Licensed Internal Code_options   : address
          control_options                  : address
          result_template                  : address
)
```

Description: The category of Licensed Internal Code options selected by the *control options* are either materialized or verified, depending on the values in the *control options*.

If the *control options* specify that Licensed Internal Code options are to be materialized, then all valid Licensed Internal Code option keywords of the specified category are materialized into the *Licensed Internal Code options* operand. If the *control options* specify that options are to be verified, then the values in the supplied *Licensed Internal Code options* operand are verified, based on the specified category. The result of the verification is returned in the *result template* operand. The types of errors that verification can detect include unrecognized option keywords and other invalid tokens. A complete list of errors that can be detected is described in Table 3 (page 716) (excluding *result subcode* zero, which means no errors were found).

The valid Licensed Internal Code options for a given category are specified with the description of that category's instruction or function, and are not further described here.

The *Licensed Internal Code options* operand specifies the address of an area that receives Licensed Internal Code options when materialized or supplies them when verified. For materialize, the maximum size of the area is specified by the *Licensed Internal Code options length* field of the *control options*, while for verify the *Licensed Internal Code options length* field specifies the length of the string to be verified.

The values materialized into or verified from the *Licensed Internal Code options* operand area are presumed to use the Unicode character set. (See *The Unicode Standard: Worldwide Character Encoding, Version 2.0*, ISBN pending.) Options are separated by commas, and some options may accept values. During verification, the case of option keywords is ignored (though the case of option values may be significant). Multiple occurrences of the same option are allowed, but mutually exclusive option keywords aren't allowed in the same string.

The special characters used in the *Licensed Internal Code options* operand are described in the following table:

Table 1. Licensed Internal Code options special characters

ASCII character(s)	Hex value	Use
comma (,)	002C	Separate options
single quote (')	0027	Contain text strings
equals sign (=)	003D	Assign values to options

ASCII character(s)	Hex value	Use
character 9	0039	Indicates numeric option value (materialize)
character A	0041	Indicates alphabetic option value (materialize)
characters 0x	00410078	Indicates the following characters should be treated as a hexadecimal numeric value (verify, create)
backslash (\)	005C	Escape character. Indicates that the following character in a character value is not to be interpreted as a special character (verify, create). With the exception of the backslash and quote characters, special characters within a quoted string do not need to be escaped.
space	0020	Ignored between tokens (verify, create)
asterisk (*)	002A	Ignored between tokens (verify, create). A keyword preceded by an asterisk means that the option was not applied when processed because the keyword was not recognized on the system where it was last processed by the associated instruction or operation. (See *Licensed Internal Code options category* for additional information.)
plus sign (+)	002B	Ignored between tokens (verify, create). A keyword preceded by a plus sign means that the option was ignored when processed because there is at least one other occurrence of the same keyword in the string. When multiple occurrences exist, the last one is the one that is applied.[2] (page 717)

The use of special characters is illustrated with the following example. Suppose a category consists of three Licensed Internal Code options, with keywords ″Keyword1″, ″Keyword2″, and ″Keyword3″. Further, suppose that ″Keyword1″ accepts no value, while ″Keyword2″ accepts a character string value, and ″Keyword3″ accepts a numeric value. Then the materialized string for this category would be ″Keyword1,Keyword2=A,Keyword3=9″. A valid input string for the same category might be ″Keyword2 =’The quick brown fox’ , KEYWORD1, keyword3= 0x1234″.

The **control options** specify which category of Licensed Internal Code options is to be selected and whether they are to be materialized or verified.

The format of the *control options* template is:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Licensed Internal Code options length	UBin(4)
4	4	Operation code	UBin(4)
		1 = Materialize Licensed Internal Code options	
		2 = Verify Licensed Internal Code options	
8	8	Licensed Internal Code options category	UBin(4)
12	C	Licensed Internal Code options sub-category	UBin(4)
16	10	Reserved (binary 0)	Char(48)
64	40	—- End —-	

The **Licensed Internal Code options length** field specifies the size (in two-byte characters) of the *Licensed Internal Code options* operand for materialize and the length of the options string (in two-byte characters) for verify. The length value cannot be greater than 65,535 or a *template value invalid* (hex 3801) exception is signalled.

The **operation code** field specifies whether a materialize or verify operation is to be performed. A value of 1 indicates that the valid options for the specified category should be materialized. A value of 2 indicates that the supplied options string should be verified, assuming the specified category. If the value of *operation code* is invalid then a *template value invalid* (hex 3801) exception is signalled.

The **Licensed Internal Code options category** field identifies the instruction or operation for which options are being materialized or verified. The valid values are shown in the table below. If the value of *Licensed Internal Code options category* is invalid then a *template value invalid* (hex 3801) exception is signalled.

The **Licensed Internal Code options subcategory** field identifies the function or subset of the above-specified instruction or operation for which options are being materialized or verified. The valid values are shown in the table below. If the value of *Licensed Internal Code options subcategory* is invalid then a *template value invalid* (hex 3801) exception is signalled.

Table 2. Licensed Internal Code options categories and sub-categories

Instruction or Operation	Category	Sub-categories
Module Creation	3	0

For the list of Licensed Internal Code options defined for category 3, see the ILE Concepts book (SC41-5606).

The format of the **result template** operand template depends on the *operation code*. For materialize, the format of the template is:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Characters materialized	UBin(4)
4	4	Reserved (binary 0)	Char(28)
32	20	—- End —-	

The **characters materialized** field returns the number of two-byte characters in the option string materialized by this instruction.

For verify, the format of the template is:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Result code	UBin(4)
		0 = Operation successful	
		1 = Error detected	
4	4	Result subcode	UBin(4)
8	8	Error offset	UBin(4)
12	C	Reserved (binary 0)	Char(20)

Offset Dec	Hex	Field Name	Data Type and Length
32	20	—- End —-	

The **result code** field returns an indication of the success of the verify operation. A value of 0 indicates that the verification was successful and no errors were found. A value of 1 indicates that errors were found during the verification.

The **result subcode** field returns a code identifying the cause of the verify failure in the event that *result code* has a value of 1.

The *result subcode* values and their meanings are listed in the following table:

Table 3. Licensed Internal Code options result subcodes

Result subcode **value**	**Meaning**
0	Verification was successful.
1	LICOPT string is ill-formed. This means that there is a syntax error in the string. It's due to an invalid character or other token occuring in the string where it doesn't belong.
2	Keyword is invalid.
3	Value has incorrect type. The value being assigned to an option has the wrong type.
4	Value is out of range. The value being assigned to an option is not in the valid range of values allowed for the option.
5	Keyword is the opposite of a prior keyword. This happens when two mutually exclusive Licensed Internal Code keywords are specified in the same string, which is invalid. Mutually exclusive keywords specify opposite options. An example of a pair of mutually exclusive keywords is BindStatic and NoBindStatic.

The **error offset** field returns the offset, in two-byte characters, to the point where the first error was detected in the options string. *Error offset* is set to 0 if *result code* indicates that verification was successful.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

32 Scalar Specification

3203 Scalar Value Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Footnotes:

[1] Mutually exclusive keywords specify opposite options. An example of a pair of mutually exclusive keywords is BindStatic and NoBindStatic.

[2] This is true in general, but there are exceptions. Some keywords can be specified multiple times in order to provide multiple pieces of information. For these, none of the keywords will be preceded by a plus sign, since they will all have been applied.

Materialize Pointer (MATPTR)

Op Code (Hex)	Operand 1	Operand 2
0512	Receiver	Pointer

Operand 1: Space pointer.

Operand 2: System pointer, space pointer data object, data pointer, instruction pointer, invocation pointer, procedure pointer, label pointer, suspend pointer, synchronization pointer, or object pointer.

Bound program access

```
Built-in number for MATPTR is 89.
MATPTR (
        receiver   : address
        pointer    : address of pointer(16)
)
```

Note

When materializing a procedure pointer, it is recommended that you use the 8-byte activation and activation group marks at the end of the procedure pointer description template. 4-byte marks can wrap and produce unexpected results.

Description: The materialized form of the pointer object referenced by operand 2 is placed in operand 1.

If the operand 2 *pointer* is a system pointer or data pointer and unresolved, the pointer is resolved before the materialization occurs.

This instruction will tolerate a damaged object referenced by operand 2 when operand 2 is a resolved pointer. The instruction will not tolerate a damaged context(s) or damaged programs when resolving pointers. Also, as a result of damage or abnormal machine termination, this instruction can indicate that an object is addressed by a context, when in fact the context will not show this as an addressed object.

A space pointer machine object cannot be specified for operand 2.

The *receiver* is a space pointer to a *materialization template*. This template must be aligned on a 16-byte boundary to materialize these types of pointers: invocation, procedure, label, and suspend. Otherwise, the *boundary alignment* (hex 0602) exception is signaled.

The format of the materialization pointed to by operand 1 is:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Pointer type	Char(1)	

Offset Dec	Offset Hex	Field Name	Data Type and Length
		Hex 01 = System pointer	
		Hex 02 = Space pointer	
		Hex 03 = Data pointer	
		Hex 04 = Instruction pointer	
		Hex 05 = Invocation pointer	
		Hex 06 = Procedure pointer	
		Hex 07 = Label pointer	
		Hex 08 = Suspend pointer	
		Hex 09 = Synchronization pointer	
		Hex 0A = Object pointer	
		Hex FF = Unsupported pointer	
9	9	Pointer description	Char(*)
*	*	—- End —-	

Pointer description depends on the *pointer type.* If *unsupported pointer* is indicated, then no other data is returned for *pointer description*. Otherwise one of the following pointer type formats is used.

Offset Dec	Offset Hex	Field Name	Data Type and Length
9	9	System pointer description	Char(68)
		The system pointer description identifies the object addressed by the pointer and the context which the object specifies as its addressing context.	
9	9	Context identification	Char(32)
9	9	Context type	Char(1
10	A	Context subtype	Char(1
11	B	Context name	Char(3
41	29	Object identification	Char(32)
41	29	Object type	Char(1
42	2A	Object subtype	Char(1
43	2B	Object name	Char(3
73	49	Pointer authorization	Char(2)
73	49	Object control	Bit 0
73	49	Object management	Bit 1
73	49	Authorization pointer	Bit 2
73	49	Space authority	Bit 3
73	49	Retrieve	Bit 4
73	49	Insert	Bit 5
73	49	Delete	Bit 6

Offset Dec	Hex	Field Name	Data Type and Length		
73	49		Update		Bit 7
73	49		Reserved (binary 0)		Bits 8-10
73	49		Execute		Bit 11
73	49		Reserved (binary 0)		Bits 12-15
75	4B		Pointer target information	Char(2)	
75	4B		Pointer target accessible from user state		Bit 0
75	4B		Reserved (binary 0)		Bits 1-15
77	4D	—- End —-			

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the materialization length exception) are signaled in the event that the receiver contains insufficient area for the materialization.

Note: If the object addressed by the system pointer specifies that it is not addressed by a context or if the context is destroyed, the **context identification** field is hex 00. If the object is addressed by the machine context, a *context type* of hex 81 is returned. No verification is made that the specified context actually addresses the object.

The following lists the **object type** codes for system object references:

Value (Hex)	Object Type
01	Access group
02	Program
03	Module
04	Context
06	Byte string space
07	Journal space
08	User profile
09	Journal port
0A	Queue
0B	Data space
0C	Data space index
0D	Cursor
0E	Index
0F	Commit block
10	Logical unit description
11	Network description
12	Controller description
13	Dump space
14	Class of service description
15	Mode description
16	Network interface description
17	Connection list

18	Queue space
19	Space
1A	Process control space
1B	Authority list
1C	Dictionary
1D	Auxiliary server
1E	Byte stream file
20	XOM object
21	Composite object group
23	Transaction control structure

Note: Only the authority currently stored in the system pointer is materialized.

If the **pointer target accessible from user state** field has a value of binary 1, then the system pointer addresses an object that is in user domain. If the *pointer target accessible from user state* field has a value of binary 0, then the system pointer addresses an object that is not in user domain.

Offset Dec	Offset Hex	Field Name	Data Type and Length
9	9	Data pointer description The data pointer description describes the current scalar and array attributes and identifies the space addressability contained in the data pointer.	Char(75)
9	9		Scalar and array attributes
9	9		Scalar type **Hex 00 =** Signed binary **Hex 01 =** Floating-point **Hex 02 =** Zoned decimal **Hex 03 =** Packed decimal **Hex 04 =** Character **Hex 06 =** Onlyns **Hex 07 =** Onlys **Hex 08 =** Either **Hex 09 =** Open **Hex 0A =** Unsigned binary
10	A		Scalar length
10	A		If binary, charac Length If zoned decima
10	A		Fractional digits
10	A		Total digits

Offset Dec	Offset Hex	Field Name	Data Type and Length	
12	C		Reserved (binary 0)	
16	10		Data pointer space addressability	
16	10		Context identification	
16	10			Context type
17	11			Context subtype
18	12			Context name
48	30		Object identification	
48	30			Object type
49	31			Object subtype
50	32			Object name
80	50		Offset into space	
84	54	—- End —-		

Note:

If the object containing the space addressed by the data pointer is not addressed by a context, the **context identification** field is hex 00. If the object is addressed by the machine context, a *context type* of hex 81 is returned.

Support for usage of a data pointer describing an Onlyns, Onlys, Either, or Open scalar value is limited. For more information, refer to the Copy Extended Characters Left Adjusted With Pad (CPYECLAP) and Set Data Pointer Attributes (SETDPAT) instructions.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
9	9	Space pointer description The space pointer description describes space addressability contained in the space pointer.	Char(79)	
9	9		Context identification	Char(32)
9	9		Context type	Char(1)
10	A		Context subtype	Char(1)
11	B		Context name	Char(30)
41	29		Object identification	Char(32)
41	29		Object type	Char(1)
42	2A		Object subtype	Char(1)
43	2B		Object name	Char(30)
73	49		Offset into space	Bin(4)
77	4D		Pointer target information	Char(2)
77	4D		Pointer target accessible from user state	Bit 0
77	4D		Pointer target is teraspace	Bit 1
77	4D		Reserved (binary 0)	Bits 2-15
79	4F		Reserved (binary 0)	Char(1)
80	50		Extended offset into space	Char(8)
88	58	—- End —-		

The **object identification** information supplied, for a space pointer which points to an implicit process space or to teraspace, is for the process control space object with which those spaces are associated.

Note: If the object associated with the space addressed by the space pointer is not addressed by a context, the **context identification** field is hex 00. If the object is addressed by the machine context, a *context type* of hex 81 is returned.

The **offset into space** field is set to a value of zero when the space pointer points to teraspace.

If the **pointer target accessible from user state** field has a value of binary 1, then the space pointer addresses a space that is in user domain and is either writeable when the thread is in user state or read only for any thread execution state. The *pointer target accessible from user state* field has a value of binary 0 otherwise.

If the **pointer target is teraspace** field has a value of binary 1, then the space pointer addresses teraspace. This field has a value of binary 0 if the space pointer addresses any other space.

The **extended offset into space** field is set whether or not the space pointer points to teraspace.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
9	9	Instruction pointer description	Char(68)	
		The instruction pointer description describes instruction addressability contained in the instruction pointer.		
9	9		Context identification	Char(32)
9	9		Context type	Char(1
10	A		Context subtype	Char(1
11	B		Context name	Char(3
41	29		Program identification	Char(32)
41	29		Program type	Char(1
42	2A		Program subtype	Char(1
43	2B		Program name	Char(3
73	49		Instruction number	Bin(4)
77	4D	—- End —-		

If the program containing the instruction currently being addressed by the instruction pointer is not addressed by a context, the *context identification* field is hex 00.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
9	9	Invocation pointer description	Char(23)	
		The invocation pointer description describes invocation addressability contained in the invocation pointer.		
9	9		Pointer status	Char(1)
9	9		Invocation no longer exists	
9	9		Pointer is from another thread	
9	9		Reserved (binary 0)	
10	A		Reserved (binary 0)	Char(6)
16	10		Containing process	System pointer
32	20	—- End —-		

Invocation no longer exists. If this field has a value of binary 1, then the invocation referenced by the pointer no longer exists.

Pointer is from another thread. If this field has a value of binary 1, then the invocation referenced by the pointer exists but belongs to a thread other than the current one.

Containing process. A system pointer to the process control space object which contains the thread to which the invocation belongs. A null pointer value is returned if the invocation no longer exists.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
9	9	Procedure pointer description	Char(71)	
		The procedure pointer description describes the activation and procedure addressability contained in the procedure pointer.		
9	9		Pointer status	Char(1)
9	9		Process object no longer exists	
9	9		Pointer is from another process	
9	9		Referenced program cannot be accessed	
9	9		Containing process owns a shared activation group	
9	9		Reserved (binary 0)	
10	A		Reserved (binary 0)	Char(6)
16	10		Module number	UBin(4)
20	14		Procedure number	UBin(4)
24	18		Activation mark	UBin(4)
28	1C		Activation group mark	UBin(4)
32	20		Containing program	System
48	30		Containing process	System
64	40		Activation mark	UBin(8)
			For Non-Bound programs, the following datatype should be used:	
64	40		Activation mark (Non-Bound program)	
72	48		Activation group mark	UBin(8)
			For Non-Bound programs, the following datatype should be used:	
72	48		Activation group mark (Non-Bound program)	
80	50	—- End —-		

Process object no longer exists. If this field has a value of binary 1, then the process object referenced by the pointer (the activation) no longer exists. All of the remaining information is returned as binary 0s.

Pointer is from another process. If this field has a value of binary 1, then the process object referenced by the pointer belongs to a process other than the current one.

Referenced program cannot be accessed. If this field has a value of binary 1, then the program referenced by the pointer could not be accessed to extract the program-related information. This may be because the program is damaged, suspended, compressed, or destroyed. The *containing program pointer*, *module number*, and *procedure number* are returned as binary 0s.

Containing process owns a shared activation group. If this field has a value of binary 1, then the process object referenced by the pointer belongs to a process that owns a shared activation group.

Module number. Index in the module list of the bound program for the module whose activation the pointer addresses.

Procedure number. Index in the procedure list of the module for the procedure addressed by the pointer.

Activation mark. The activation mark of the activation that contains the activated procedure. Zero if the program activation no longer exists. The value returned in the 4-byte activation mark may have wrapped.

Activation group mark. An activation group mark of the activation group that contains the activated procedure. Zero if the program activation no longer exists. The value returned in the 4-byte activation group mark may have wrapped.

Containing program. A system pointer to the program object that contains the procedure. Null if the program activation no longer exists.

Containing process. A system pointer to the process control space object which contains the procedure's activation group. A null pointer value is returned if the process control space object no longer exists, or if it is no longer possible to determine the containing process for a destroyed activation group.

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
9	9	Label pointer description	Char(*)
		The label pointer description describes instruction addressability contained in the label pointer.	
9	9		Pointer status
9	9		Reserved (binary 0)
9	9		Referenced program is damaged, suspended, compresse
9	9		Reserved (binary 0)
10	A		Reserved (binary 0)
16	10		Module number
20	14		Procedure number
24	18		Number of statement IDs
28	1C		Internal identifier
32	20		Containing program
48	30		Statement ID
*	*	—- End —-	

Referenced program is damaged, suspended, compressed, or destroyed. If this field has a value of binary 1, then the program referenced by the pointer could not be accessed to extract the remaining information. The remainder of the template is binary 0s with the exception of the *containing program* pointer, which will be binary 0s if the program has been destroyed or so seriously damaged that its identity cannot be determined.

Module number. Index in the module list of the bound program for the module containing the label.

Procedure number. Index in the procedure list of the module for the procedure containing the label.

Number of statement IDs. Number of entries in the statement ID list. (Multiple statement IDs may be associated with a single location in the created program due to optimizations that combine similar code sequences.)

Internal identifier. A machine-dependent value which identifies the label relative to the internal structure of the program. For use by service personnel.

Containing program. A system pointer to the program object that contains the label.

Statement ID. Each statement ID is a compiler-supplied unsigned Bin(4) number which allows the compiler to identify the source statement associated with a particular sequence of instructions.

Offset Dec	Offset Hex	Field Name	Data Type and Length
9	9	Suspend pointer description The suspend pointer description describes instruction addressability contained in the suspend pointer.	Char(*)
9	9		Pointer status
9	9		Reserved (binary 0)
9	9		Reserved (binary 0)
9	9		Referenced program is damaged, suspended, compressed o
9	9		Reserved (binary 0)
10	A		Reserved (binary 0)
16	10		Module number
20	14		Procedure number
24	18		Number of statement IDs
28	1C		Internal identifier
32	20		Containing program
48	30		Statement ID
*	*	—- End —-	

Referenced program is damaged, suspended, compressed, or destroyed. If this field has a value of binary 1, then the program referenced by the pointer could not be accessed to extract the remaining information. The remainder of the template is binary 0s with the exception of the *containing program* pointer, which will be binary 0s if the program has been destroyed or so seriously damaged that its identity cannot be determined.

Module number. Index in the module list of the bound program for the module containing the suspend point.

Procedure number. Index in the procedure list of the module for the procedure containing the suspend point.

Number of statement IDs. Number of entries in the statement ID list. (Multiple statement IDs may be associated with a single location in the created program due to optimizations that combine similar code sequences.)

Internal identifier. A machine-dependent value which locates the suspend point relative to the internal structure of the program. For use by service personnel.

Containing program. A system pointer to the program object that contains the suspend point.

Statement ID. Each statement ID is a compiler-supplied unsigned Bin(4) number which allows the compiler to identify the source statement associated with a particular sequence of MI instructions.

Note: For suspend pointers which address non-bound programs, module number and procedure number are returned as binary 0s, and the statement ID list is returned with one value which is the MI instruction number of the suspend point.

Offset Dec	Offset Hex	Field Name	Data Type and Length
9	9	Synchronization pointer description The synchronization pointer description describes the object addressability contained in the synchronization pointer.	Char(4)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
9	9		Pointer status	Char(1)
9	9		Synchronization object no longer exists	Bit 0
9	9		Reserved (binary 0)	Bits 1-7
10	A		Synchronization object type	Char(2)
			Hex 0000 = Synchronization object no longer exists	
			Hex 0001 = Mutex	
			Hex 0002 = Semaphore	
12	C		Reserved (binary 0)	Char(1)
13	D	—- End —-		

Synchronization object no longer exists. If this field has a value of binary 1, then the synchronization object referenced by the pointer no longer exists. All of the remaining information is returned as binary 0s.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
9	9	Object pointer description The object pointer description describes the object identification contained in the object pointer.	Char(16)	
9	9		Object ID	Char(12)
21	15		Reserved (binary 0)	Char(4)
25	19	—- End —-		

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Pointer Information (MATPTRIF)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0517	Receiver	Pointer	Selection mask

Operand 1: Space pointer.

Operand 2: Suspend pointer, system pointer or space pointer data object.

Operand 3: Character(4) scalar.

Bound program access

```
Built-in number for MATPTRIF is 420.
MATPTRIF (
          receiver         : address
          pointer          : address of pointer(16)
          selection_mask   : address
)
```

Description: The attributes selected with operand 3 of the pointer object identified by operand 2 are materialized into the *receiver* identified by operand 1.

Operand 1:

The *receiver* is a space pointer to a *materialization template*. This template must be aligned on a 16-byte boundary, otherwise the *boundary alignment* (hex 0602) exception is signaled. If any of the reserved fields in the template are not zero, a *template value invalid* (hex 3801) exception will be signaled.

Some of the fields in the *materialization template* are input to the instruction and remain unchanged by the instruction. Input fields are indicated in the description of the template that follows, and are used to control the amount of information to be materialized. The remaining fields in the materialization template are output from the instruction.

If this instruction ends abnormally, the contents of the materialization template are undefined.

The **materialization template** has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization template	Char(*)	
0	0		Template size specification	Char(8)
0	0		Number of bytes provided (input)	Bin(4)
4	4		Number of bytes available	Bin(4)
8	8		Reserved	Char(7)
15	F		Pointer type	Char(1)

Offset Dec	Hex	Field Name	Data Type and Length	
			Hex 01 = System pointer	
			Hex 02 = Space pointer	
			Hex 08 = Suspend pointer	
16	10		Pointer description	Char(*)
*	*	—- End —-		

Template size specification

This field contains size information about the materialization. The number of bytes in the materialization is the lesser of the *number of bytes provided* and the *number of bytes available*.

Number of bytes provided

This input field is the number of bytes in the *materialization template* provided for the materialization. It must have a value of eight or more, otherwise the *materialization length invalid* (hex 3803) exception is signaled.

Number of bytes available

This output field is the number of bytes in the available materialization. If the materialization template is larger than the available materialization, the excess bytes in the template are unchanged. If the template is smaller than the available materialization, no exceptions are signaled and as many bytes as can be contained in the template are materialized.

Pointer type

This output field indicates the type of the pointer identified by the operand 2 *pointer*.

Pointer description

This field contains both input and output subfields. Each output field contains a materialized attribute of the operand 2 *pointer* only if selected with the operand 3 *selection mask*. Otherwise, the output field remains unchanged. The format of the pointer description is determined by the *pointer type* of the operand 2 *pointer*.

The *pointer description* for a *suspend pointer* has the following format:

Offset Dec	Hex	Field Name	Data Type and Length	
16	10	Suspend pointer description	Char(192)	
16	10		Reserved	Char(1)
17	11		Program type	Char(1)
			Hex 00 = Non-bound program	
			Hex 01 = Bound program	
			Hex 02 = Bound service program	
			Hex 04 = Java(TM) program	
18	12		Program CCSID	Char(2)
20	14		Program name	Char(30)
50	32		Program context name	Char(30)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
80	50		Reserved	Char(4)
84	54		Module name	Char(30)
114	72		Module qualifier name	Char(30)
144	90		Reserved	Char(4)
148	94		Procedure dictionary ID	Bin(4)
152	98		Length of procedure name requested (input)	Bin(4)
156	9C		Length of procedure name available	Bin(4)
160	A0		Pointer to procedure name (input)	Space pointer
176	B0		Reserved	Char(8)
184	B8		Number of statement IDs requested (input)	Bin(4)
188	BC		Number of statement IDs available	Bin(4)
192	C0		Pointer to statement IDs (input)	Space pointer
208	D0	—- End —-		

Program type
This output field indicates the Program Model of a program object, which is determined by how the program was created. This field is necessary since the object type and object subtype do not provide enough information to identify the Program Model of a program object. Knowing the program type is useful in selecting appropriate program specific instructions. For this instruction, it is useful in determining whether several fields in this materialization template are valid, as indicated in the description of each field.

Program CCSID
This output field is the *coded character set identifier* of the bound program having *program name*. This field is not valid if *program type = hex 00*.

Program name
This output field is the name of the program object whose invocation contains the suspend point.

Program context name
This output field is the name of the context in which the program having *program name* resides. This field contains hex zeros if the program does not reside in a context.

Module name
This output field is the name of the module which contained the definition of the procedure identified by *procedure dictionary ID* at the time the program having *program name* was created. This field is not valid if *program type = hex 00*.

Module qualifier name
This output field is the module qualifier, used to differentiate between modules having the same *module name*. It was provided when the program having *program name* was created. This field is not valid if *program type = hex 00*.

Procedure dictionary ID
This output field is the dictionary ID of the procedure containing the suspend point in the invocation of the program object named *program name*. This field is not valid if *program type = hex 00*.

Length of procedure name requested
This input field is the number of characters in *procedure name* provided for the materialization. If the length requested is zero, then the *pointer to procedure name* need not be supplied and will remain unchanged. This field is ignored if *program type = hex 00*.

Length of procedure name available
This output field is the number of characters in the **available procedure name**. This field is not valid if *program type = hex 00*.

Pointer to procedure name
This input field is a pointer to the space provided for the materialized name of the procedure identified by *procedure dictionary ID*. This field need not be supplied and remains unchanged if the *length of procedure name requested* is zero. This field is not valid if *program type = hex 00*.

The **procedure name** has the following format:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Procedure name	Char(*)
*	*	—- End —-	

Procedure name
This output field is the materialized name of the procedure where the number of characters in the materialized name is the lesser of the *length of procedure name requested* and the *length of procedure name available*. If this field is larger than the *available procedure name*, the excess characters in the field are unchanged. If this field is smaller than the *available procedure name*, no exceptions are signaled and as many characters as can be contained in the field are materialized. This field remains unchanged if *length of procedure name requested* is zero. This field is not valid if *program type = hex 00*.

Number of statement IDs requested
This input field is the number of elements in the array *statement IDs* provided for the materialization. If the number requested is zero, then the *pointer to statement IDs* need not be supplied and will remain unchanged.

Number of statement IDs available
This output field is the number of elements in the array of **available statement IDs**.

Pointer to statement IDs
This input field is a pointer to the space provided for the materialized statement IDs associated with this suspend point. (Multiple statement IDs may be associated with a single location in the created program due to optimizations that combine similar code sequences). This field need not be supplied and remains unchanged if the *number of statement IDs requested* is zero.

The **statement IDs** have the following format:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Statement IDs	[*] Bin(4)
*	*	—- End —-	

Statement IDs
This output field is an array of materialized statement IDs associated with this suspend point, where the number of elements in the array is the lesser of the *number of statement IDs requested* and the *number of statement IDs available*. Each statement ID was previously supplied by the compiler and identifies a source statement associated with the suspend point. If this field is larger than the array of *available statement IDs*, the excess array elements in the field are unchanged. If this field is smaller than the array of *available statement IDs*, no exceptions are

signaled and as many IDs as can be contained in the array are materialized. This field remains unchanged if *number of statement IDs requested* is zero.

The *pointer description* for a *system pointer* or *space pointer* and a selection mask with an *information option* of 0 has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	ASP number description	Char(2)	
16	10		ASP number	UBin(2)
18	12	—- End —-		

ASP number
This output field contains the ASP number assigned by the machine to the ASP that contains the storage that is the target of the pointer. If the pointer is a pointer to teraspace storage, the *ASP number* returned is 1 (ie, the system ASP). This instruction is allowed in all ASP LUD states but, if the state is not varyon or active, an *object not available* (hex 220B) exception may be signalled. If the ASP does not have a LUD, or if it does and the LUD state is varyon or active, an *object not available* (hex 220B) exception is not signalled.

Operand 2:

The *pointer* is the pointer object to be materialized.

Operand 3:

The *selection mask* is used to select which attributes of the operand 2 *pointer* are to be materialized. The format of the *selection mask* is determined by the *pointer type* of the operand 2 *pointer*.

For *suspend pointers*, the *selection mask* is a bit mask that is used to select which attributes of the pointer are to be materialized. If the bit has a value of binary 1, then the attribute is materialized into the associated *pointer description* output field in the operand 1 *materialization template*. If the bit has a value of binary 0, the output field remains unchanged. If any of the reserved bits in the mask are not zero, a *scalar value invalid* (hex 3203) exception is signalled.

The *selection mask* for a *suspend pointer* has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Suspend pointer selection mask	Char(4)	
0	0		Reserved	Bit 0
0	0		Program type	Bit 1
			0 = Do not materialize	
			1 = Materialize *program type* field	
0	0		Program CCSID	Bit 2
			0 = Do not materialize	
			1 = Materialize *program CCSID* field	
0	0		Program name	Bit 3
			0 = Do not materialize	
			1 = Materialize *program name* field	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0		Program context name	Bit 4
			0 = Do not materialize	
			1 = Materialize *program context name* field	
0	0		Reserved	Bit 5
0	0		Module name	Bit 6
			0 = Do not materialize	
			1 = Materialize *module name* field	
0	0		Module qualifier name	Bit 7
			0 = Do not materialize	
			1 = Materialize *module qualifier name* field	
0	0		Reserved	Bit 8
0	0		Procedure dictionary ID	Bit 9
			0 = Do not materialize	
			1 = Materialize *procedure dictionary ID* field	
0	0		Procedure name	Bit 10
			0 = Do not materialize	
			1 = Materialize *length of procedure name available* field and the *procedure name* addressed by the *pointer to procedure name* field	
0	0		Reserved	Bit 11
0	0		Statement IDs	Bit 12
			0 = Do not materialize	
			1 = Materialize *number of statement IDs available* field and the *statement IDs* addressed by the *pointer to statement IDs* field	
0	0		Reserved	Bits 13-31
4	4	—- End —-		

The *selection mask* for a *system pointer* or *space pointer* has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Selection mask	Char(4)	
0	0		Information option	UBin(2)
			0 = Materialize ASP number	
2	2		Reserved	Char(2)
4	4	—- End —-		

Warning: Temporary Level 3 Header

Authorization Required

•
- Retrieve
  –
  – Program object
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•
- Materialize
  –
  – Program object
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Pointer Locations (MATPTRL)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0513	Receiver	Source	Length

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Binary scalar.

Bound program access
Built-in number for MATPTRL is 90. MATPTRL (receiver : address source : address length : address of signed binary(4))

Description: This instruction finds the 16-byte pointers in a subset of a space and produces a bit mapping of their relative locations.

The area addressed by the operand 2 space pointer is scanned for a length equal to that specified in operand 3. A bit in operand 1 is set for each 16 bytes of operand 2. The bit is set to binary 1 if a pointer exists in the operand 2 space, or the bit is set to binary 0 if no pointer exists in the operand 2 space.

Operand 1 is a space pointer addressing the *receiver* area. One bit of the *receiver* is used for each 16 bytes specified by operand 3. If operand 3 is not a 16-byte multiple, then the bit position in operand 1 that corresponds to the last (odd) bytes of operand 2 is set to 0. Bits are set from left to right (bit 0, bit 1,...) in operand 1 as 16-byte areas are interrogated from left to right in operand 2. The number of bits set in the *receiver* is always a multiple of 8. Those rightmost bits positions that do not have a corresponding area in operand 2 are set to 0.

The format of the operand 1 *receiver* is:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Template size specification	Char(8)	
0	0		Number of bytes provided	Bin(4)
4	4		Number of bytes available	Bin(4)
8	8	Pointer locations	Char(*)	
*	*	—- End —-		

Operand 2 must address a 16-byte aligned area; otherwise, a *boundary alignment* (hex 0602) exception is signaled. If the value specified by operand 3 is not positive, the *scalar value invalid* (hex 3203) exception is signaled.

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the *receiver* contains insufficient area for materialization.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Process Activation Groups (MATPRAGP)

Op Code (Hex)	Operand 1
MATPRAGP2 0339	Receiver
MATPRAGP 0331	Receiver

Operand 1: Space pointer.

Bound program access
Built-in number for MATPRAGP2 is 662. MATPRAGP2 (receiver : address) Built-in number for MATPRAGP is 123. MATPRAGP (receiver : address)

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note
It is recommended that you use the MATPRAGP2 instruction which supports 8-byte activation group marks. 4-byte marks can wrap and produce unexpected results.

Description: This instruction provides a list of the activation groups which exist in the current process. Operand 1 locates a template which receives information.

The materialization template identified by operand 1 must be 16-byte aligned in the space. This materialization template has the following format:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Number of bytes provided for materialization	Bin(4)
4	4	Number of bytes available for materialization	Bin(4)

Offset Dec	Offset Hex	Field Name	Data Type and Length
8	8	Activation group count	Bin(4)
12	C	Activation group list	Char(*)
*	*	—- End —-	

The activation group list format is different for MATPRAGP and MATPRAGP2.

Format for MATPRAGP2 activation group list:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
12	C	Reserved	Char(4)	
16	10	Activation group list	[*] Char(8)	
		(repeated *activation group count* times)		
16	10		Activation group marks	UBin(8)
			For Non-Bound programs, the following datatype should be used:	
16	10		Activation group marks (Non-Bound program)	Char
*	*	—- End —-		

Format for MATPRAGP activation group list:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
12	C	Activation group list	[*] Char(4)	
		(repeated *activation group count* times)		
12	C		Activation group marks	UBin(4)
*	*	—- End —-		

The Materialize Activation Group Attributes instruction can be used to examine the attributes of an individual activation group.

The first 4 bytes of the materialization template specify the **number of bytes provided** for use by the instruction. In all cases if the number of bytes provided is less than 8 then a *materialization length invalid* (hex 3803) exception will be signaled.

The second 4 bytes of the instruction indicate the actual **number of bytes available** to be returned. In no case does the instruction return more bytes of information than those available.

Activation group count
This is the number of activation groups within the process. It is also the extent of the activation group list which follows.

Activation group list
This is the list of activation groups which exist within the current process.

Activation group marks
This is an array of activation group mark values. Each entry denotes an activation group currently existent within the process. The value returned in the 4-byte activation group mark may have wrapped.

Warning: Temporary Level 3 Header

Authorization

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Process Attributes (MATPRATR)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0333	Receiver	Process control space	Materialization options

Operand 1: Space pointer.

Operand 2: System pointer or null.

Operand 3: Character(1) scalar.

Bound program access

```
Built-in number for MATPRATR is 65.
MATPRATR (
          receiver                  : address
          process_control_space     : address of system pointer OR
                                      null operand
          materialization_options   : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The instruction causes either one specific attribute or all the attributes of the designated process to be materialized.

Operand 1 specifies a space that is to receive the materialized attribute values. The space pointer specified in operand 1 must address a 16-byte aligned area.

Operand 2 identifies the process control space associated with the process whose attributes are to be materialized. If operand 2 is **NULL**, the attributes being materialized will be those associated with the thread that issued the instruction. Otherwise, the attributes being materialized will be those associated with the initial thread of the identified process. If process attributes are being materialized by a thread in another process, the thread must be contained in the process that initiated the subject process or the thread must have process control special authorization defined in its user profile or in a currently adopted user profile.

Operand 3 specifies which process attribute is to be materialized.

A summary of the allowable hex values for operand 3 follows.

Table 1. Materialize Process Attributes Scope

Option on Materialize Process Attributes	Page	Attribute Scope	
		Valid Process Control Space pointer	Null Operand
00 - Entire PDT	reference #1 (page 745)	Process and initial thread	Process and issuing thread
01 - Process type	reference #1 (page 745)	Process	Process
02 - Instruction wait access state control	reference #1 (page 745)	Process	Process
03 - Time slice end access state control	reference #1 (page 745)	Process	Process
04 - Time slice event option	reference #1 (page 745)	Process	Process

Option on Materialize Process Attributes	Page	Attribute Scope	
		Valid Process Control Space pointer	Null Operand
06 - Initiation phase program option	reference #1 (page 745)	Process	Process
07 - Problem phase program option	reference #1 (page 745)	Process	Process
08 - Termination phase program option	reference #1 (page 745)	Process	Process
09 - Process default exception handler option	reference #1 (page 745)	Process	Process
0A - Name resolution list option	reference #1 (page 745)	Initial thread	Issuing thread
0B - Process access group option	reference #1 (page 745)	Process	Process
0C - Signal event control mask	reference #2 (page 749)	Process	Process
0D - Number of event monitors	reference #3 (page 749)	Process	Process
0E - Process Priority	reference #4 (page 749)	Process	Process
0F - Main storage pool ID	reference #5 (page 749)	Initial thread	Issuing thread
10 - Maximum temporary auxiliary storage allowed	reference #6 (page 749)	Process	Process
11 - Time slice interval	reference #7 (page 749)	Process	Process
12 - Default time-out interval	reference #8 (page 750)	Process	Process
13 - Maximum processor time allowed	reference #9 (page 750)	Process	Process
14 - Multi-programming level class ID	reference #10 (page 750)	Initial thread	Issuing thread
15 - Modification control indicators	reference #11 (page 750)	Process	Process
16 - User profile pointer	reference #12 (page 751)	Initial thread	Issuing thread
17 - Process Communications Object (PCO) pointer	reference #13 (page 751)	Process	Process
18 - Name resolution list pointer	reference #14 (page 751)	Process	Process
19 - Initiation phase program pointer	reference #15 (page 752)	Process	Process
1A - Termination phase program pointer	reference #16 (page 752)	Process	Process
1B - Problem phase program pointer	reference #17 (page 752)	Process	Process
1C - Process default exception handler program pointer	reference #18 (page 752)	Process	Process
1F - Process access group pointer	reference #19 (page 752)	Process	Process
20 - Process status indicators	reference #20 (page 752)	Process and initial thread	Process and issuing thread
21 - Process resource usage attributes	reference #21 (page 756)	Process	Process

Option on Materialize Process Attributes	Page	Attribute Scope	
		Valid Process Control Space pointer	Null Operand
22 - Obsolete	reference #22 (page 757)	Not applicable	Not applicable
23 - Thread performance attributes	reference #23 (page 757)	Initial thread	Issuing thread
24 - Execution status attributes	reference #24 (page 758)	Process and initial thread	Process and issuing thread
25 - Process control space pointer	reference #25 (page 760)	Process	Process
26 - Group profile list	reference #26 (page 760)	Initial thread	Issuing thread
27 - Group profile list option	reference #27 (page 761)	Initial thread	Issuing thread
28 - Process category	reference #28 (page 761)	Process	Process
29 - Queue space object pointer	reference #29 (page 761)	Process	Process
2A - Secondary process communications object (PPCO) pointer	reference #30 (page 761)	Process	Process
2B - Signal enablement option	reference #31 (page 761)	Process	Process
2C - Process signal controls	reference #32 (page 761)	Process and initial thread	Process and issuing thread
2D - Lock statistics	reference #33 (page 763)	Initial thread	Issuing thread

The value of attribute scope is as follows:

Attribute Scope	Meaning
Process	The attribute is maintained as the process level resource. The value materialized has the same value for all threads within the process.
Initial thread	The attribute is maintained as a thread level resource. The value materialized is only applicable to the initial thread within the process.
Issuing thread	The attribute is maintained as a thread level resource. The value materialized is only applicable to the thread within the process that is being materialized.
Process and initial thread	The attribute being materialized is derived from values maintained at the process level and the initial thread within the process.
Process and issuing thread	The attribute being materialized is derived from values maintained at the process level and the issuing thread within the process.

The materialization template has the following general format when a scalar attribute is materialized:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Process scalar attributes	Char(*)	
*	*	—- End —-		

The materialization template has the following general format when a pointer attribute is materialized:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Reserved (binary 0)	Char(8)	
16	10	Process pointer attribute	System pointer or Space pointer	
32	20	—- End —-		

The following attributes require materialization sizes of varying lengths. The attributes to be materialized and their operand 3 materialization option values follow:

•

- Process control attributes

  This template is returned for the following operand 3 values:
  - Hex 01 - Process type
  - Hex 02 - Instruction wait access state control
  - Hex 03 - Time slice end access state control
  - Hex 04 - Time slice event option
  - Hex 06 - Initiation phase program option
  - Hex 07 - Problem phase program option
  - Hex 08 - Termination phase program option
  - Hex 09 - Process default exception handler option
  - Hex 0A - Name resolution list option
  - Hex 0B - Process access group option
  - Hex 27 - Group profile list option
  - Hex 2B - Signal enablement option

Offset Dec	Offset Hex	Field Name	Data Type and Length	
8	8	Process control attributes	Char(4)	
8	8		Process type	Bit 0
			1 = Independent process	
8	8		Instruction wait access state control	Bit 1
			0 = Access state modification is not allowed	
			1 = Access state modification is allowed if specified	
8	8		Time slice end access state control	Bit 2
			0 = Access state modification is not allowed	
			1 = Access state modification is allowed if specified	
8	8		Time slice end event option	Bit 3
			0 = Time slice expired without entering instruction wait event is not signaled	
			1 = Time slice expired without entering instruction wait event is signaled	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
8	8		Reserved (binary 0)	Bit 4
8	8		Initiation phase program option	Bit 5
			0 = No initiation phase program specified (do not enter initiation phase)	
			1 = Initiation phase program specified (enter initiation phase)	
8	8		Problem phase program option	Bit 6
			0 = No problem phase program specified (do not enter problem phase)	
			1 = Problem phase program specified (enter problem phase)	
8	8		Termination phase program option	Bit 7
			0 = No termination phase program specified (do not enter termination phase)	
			1 = Termination phase program specified (enter termination phase)	
8	8		Process default exception handler option	Bit 8
			0 = No process default exception handler	
			1 = Process default exception handler specified	
8	8		Name resolution list option	Bit 9
			0 = No name resolution list specified	
			1 = Name resolution list specified	
8	8		Process access group option	Bit 10
			0 = No process access group specified	
			1 = Process access group specified	
8	8		Group profile list option	Bit 11
			0 = No group profile list specified	
			1 = Group profile list specified	
8	8		Process category specified	Bit 12
			0 = No process category specified when the process was initiated	
			1 = A process category was specified when the process was initiated	
8	8		Recycling control for process storage addresses used by user state programs	Bit 13

Offset Dec	Offset Hex	Field Name	Data Type and Length
			0 = Process storage addresses used by user state programs are not recycled within the process
			1 = Process storage addresses used by user state programs are recycled within the process
8	8		Initial thread automatic storage access group membership control — Bit 14
			0 = The machine is free to create automatic storage areas for the initial thread within the process access group
			Note: This does not guarantee that the automatic areas will in fact be access group members. This is an advisory flag informing the machine that the MI user would prefer these areas to be access group members.
			1 = Initial thread automatic storage areas will not be created within the process access group
8	8		Implicitly created activation group's static storage access group membership control — Bit 15
			0 = The machine is free to create static storage areas of implicitly created activation groups within the process access group
			Note: This does not guarantee that the static areas will in fact be access group members. This is an advisory flag informing the machine that the MI user would prefer these areas to be access group members.
			1 = Implicitly created activation group's static storage areas will not be created within the process access group
8	8		Implicitly created activation group's default heap storage access group membership control — Bit 16

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = The machine is free to create default heap storage areas of implicitly created activation groups within the process access group **Note:** This does not guarantee that the activation group default heap areas will in fact be access group members. This is an advisory flag informing the machine that the MI user would prefer these areas to be access group members.	
			1 = Implicitly created activation group's default heap storage areas will not be created within the process access group	
8	8		Template extension present	Bit 17
			0 = The process was initiated using a PDT which did not contain a template extension area	
			1 = The process was initiated using a PDT which did contain a template extension area	
8	8		Signal enablement option	Bit 18
			0 = The process is not enabled for signals	
			1 = The process is enabled for signals	
8	8		Threads enablement option	Bit 19
			0 = The process can not have secondary threads that have a *thread type* of *user thread*	
			1 = The process is enabled for secondary threads	
8	8		Secondary threads control option	Bit 20
			0 = Do not allow initiation of secondary threads in the process	
			1 = Allow initiation of secondary threads in the process	
8	8		Termination phase program event mask option	Bit 21
			0 = Do not change the event mask state of the initial thread when the termination phase program is invoked in the initial thread.	
			1 = Mask the initial thread for events when the termination phase program is invoked in the initial thread.	
8	8		Reserved (binary 0)	Bits 22-31
12	C	—- End —-		

The resource management attributes and data types are as follows:

•

- Hex 0C = Signal event control mask

Offset Dec	Hex	Field Name	Data Type and Length
8	8	Signal event control mask	Char(2)
10	A	—- End —-	

- Hex 0D = Number of event monitors

Offset Dec	Hex	Field Name	Data Type and Length
8	8	Number of event monitors	Bin(2)
10	A	—- End —-	

- Hex 0E = Process priority

Offset Dec	Hex	Field Name	Data Type and Length
8	8	Process priority	Char(1)
9	9	—- End —-	

- Hex 0F = Main storage pool ID

Offset Dec	Hex	Field Name	Data Type and Length
8	8	Main storage pool ID	Char(1)
9	9	—- End —-	

The **main storage pool ID** materialized is the value that is currently in effect for the thread. Its value is either the *main storage pool ID* for the process or the *time-slice-end main storage pool ID* for the process. The materialization option hex 00 materializes the current values for both main storage pool IDs.

- Hex 10 = Maximum temporary auxiliary storage allowed

Offset Dec	Hex	Field Name	Data Type and Length
8	8	Maximum temporary auxiliary storage allowed in bytes	Bin(4)
12	C	Maximum temporary auxiliary storage allowed in megabytes	UBin(4)
16	10	—- End —-	

A process which is allowed to have more than 2,147,483,647 bytes of temporary auxiliary storage will have 2,147,483,647 returned for **maximum temporary auxiliary storage allowed in bytes**. A process which can have an unlimited amount of temporary auxiliary storage, will have 2,147,483,647 returned for the *maximum temporary auxiliary storage allowed in bytes* and 0 returned for the *maximum temporary auxiliary storage allowed in megabytes*.

- Hex 11 = Time slice interval

Offset Dec	Hex	Field Name	Data Type and Length
8	8	Time slice interval	Char(8)
16	10	—- End —-	

- Hex 12 = Default time-out interval

Offset Dec	Offset Hex	Field Name	Data Type and Length
8	8	Default time-out interval	Char(8)
16	10	—- End —-	

- Hex 13 = Maximum processor time allowed

Offset Dec	Offset Hex	Field Name	Data Type and Length
8	8	Maximum processor time allowed	Char(8)
16	10	—- End —-	

- Hex 14 = Multi-programming level class ID

Offset Dec	Offset Hex	Field Name	Data Type and Length
8	8	Multi-programming level class ID	Char(1)
9	9	—- End —-	

The **multi-programming level class ID** materialized is the value that is currently in effect for the thread. Its value is either the *multi-programming level class ID* for the process or the *time-slice-end multi-programming level class ID* for the process. The materialization option hex 00 materializes the current values for both multi-programming level class IDs.

- Hex 15 = Modification control indicators

  The **modification control indicators** are materialized when the operand 3 value is hex 15. Each indicator specifies the modification options allowed for a process either by a thread within the process, by a thread in the initiating process or by a thread in a process whose governing user profile(s) has process control special authorization. The possible values of each *modification control indicator* are as follows:

00 =	Modification of the attribute is not allowed.
01 =	Modification is allowed only in the initiation and termination phases, and only by threads within the executing process. Threads within processes other than the process being modified cannot modify this attribute.
11 =	Modification is allowed in all phases and by threads within all processes.

The bit assignments of the *modification control indicators* are as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
8	8	Modification control indicators	Char(8)	
8	8		Instruction wait access state control	Bits 0-1
8	8		Time slice end access state control	Bits 2-3
8	8		Time slice event option	Bits 4-5
8	8		Reserved (binary 0)	Bits 6-7
8	8		Problem phase program option	Bits 8-9
8	8		Termination phase program option	Bits 10-11
8	8		Process default exception handler option	Bits 12-13
8	8		Name resolution list option	Bits 14-15
8	8		Signal event control mask	Bits 16-17
8	8		Process priority	Bits 18-19

Offset Dec	Hex	Field Name	Data Type and Length	
8	8		Main storage pool ID	Bits 20-21
8	8		Maximum temporary auxiliary storage allowed	Bits 22-23
8	8		Time slice interval	Bits 24-25
8	8		Default time-out interval	Bits 26-27
8	8		Maximum processor time allowed	Bits 28-29
8	8		Multi-programming level class ID	Bits 30-31
8	8		User profile pointer	Bits 32-33
8	8		Reserved (binary 0)	Bits 34-35
8	8		Name resolution list pointer	Bits 36-37
8	8		Termination phase program pointer	Bits 38-39
8	8		Problem phase program pointer	Bits 40-41
8	8		Process default exception handler	Bits 42-43
8	8		Group profile list	Bits 44-45
8	8		Obsolete	Bits 46-47
8	8		Process category	Bits 48-49
8	8		Recycling control for process storage addresses used by user state programs	Bits 50-51
8	8		Signal enablement option	Bits 52-53
8	8		Process signals controls	Bits 54-55
8	8		Secondary threads option	Bits 56-57
8	8		Reserved (binary 0)	Bits 58-63
16	10	—- End —-		

The process pointer attributes which may be materialized are the following:

•

• Hex 16 = User profile pointer

Offset Dec	Hex	Field Name	Data Type and Length
16	10	User profile pointer	System pointer
32	20	—- End —-	

The system pointer with addressability to the effective user profile is placed into the space addressed by operand 1. If the materialization option hex 00 is specified in operand 3, a reserved Char(7) field is included at this point, prior to the materialized user profile pointer.

• *(Ref #13.)* Hex 17 = Process communication object (PCO) pointer

Offset Dec	Hex	Field Name	Data Type and Length
16	10	Process communication object pointer	Space pointer
32	20	—- End —-	

The PCO space pointer is placed in the space addressed by operand 1.

• *(Ref #14.)* Hex 18 = Name resolution list (NRL) pointer

Offset Dec	Hex	Field Name	Data Type and Length
16	10	Name resolution list pointer	Space pointer
32	20	—- End —-	

The space pointer to the NRL is placed in the space addressed by operand 1.

- Hex 19 = Initiation phase program pointer

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Initiation phase program pointer	System pointer
32	20	—- End —-	

The system pointer to the program is placed in the space addressed by operand 1.

- Hex 1A = Termination phase program pointer

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Termination phase program pointer	System pointer
32	20	—- End —-	

The system pointer to the program is placed in the space addressed by operand 1.

- Hex 1B = Problem phase program pointer

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Problem phase program pointer	System pointer
32	20	—- End —-	

The system pointer to the program is placed in the space addressed by operand 1.

- Hex 1C = Process default exception handler (PDEH) program

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Process default exception handler program	System pointer
32	20	—- End —-	

The system pointer to the PDEH is placed in the space addressed by operand 1.

- Hex 1F = Process access group (PAG)

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Process access group	System pointer
32	20	—- End —-	

The system pointer with addressability to the PAG is placed in the space addressed by operand 1.

The following attributes require materialization sizes of varying lengths. The attributes to be materialized and their operand 3 materialization option values follow.

-

- Hex 20 = Process status indicators

  Process status indicators are materialized when the value of operand 3 is hex 20. The format and associated values of this attribute are as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
8	8	Thread state	Char(2)	
8	8		External existence state	Bits 0-2

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			000 = Suspended due to Suspend Process or Suspend Thread	
			010 = Suspended due to Suspend Process or Suspend Thread, in instruction wait	
			100 = Active, in ineligible wait	
			101 = Active, in current MPL	
			110 = Active, in instruction wait	
			111 = Active, in instruction wait, in current MPL	
8	8		Invocation exit active	Bit 3
8	8		Stopped by a signal	Bit 4
8	8		Suspended by Suspend Thread	Bit 5
8	8		Reserved (binary 0)	Bits 6-7
8	8		Internal processing phase	Bits 8-10
			001 = Initiation phase	
			010 = Problem phase	
			100 = Termination phase	
8	8		Reserved (binary 0)	Bits 11-15
10	A	Pending thread interrupts	Char(2)	
10	A		Time slice end	Bit 0
10	A		Transfer lock	Bit 1
10	A		Asynchronous lock retry	Bit 2
10	A		Suspend process	Bit 3
10	A		Resume process	Bit 4
10	A		Modify resource management attribute	Bit 5
10	A		Modify process or thread attribute	Bit 6
10	A		Terminate machine processing	Bit 7
10	A		Terminate process or thread	Bit 8
10	A		Wait time-out	Bit 9
10	A		Event schedule	Bit 10
10	A		Thread operations between threads	Bit 11
10	A		Cancel long running instruction	Bit 12
10	A		Reserved (binary 0)	Bit 13
10	A		Deliver queue space message	Bit 14
10	A		Signal schedule	Bit 15
12	C	Process initial internal termination status	Char(3)	
12	C		Initial internal termination reason	Bits 0-7

Offset Dec	Hex	Field Name	Data Type and Length	
			Hex 80 = Return from first invocation in problem phase	
			Hex 40 = Return from first invocation in initiation phase and no problem phase program specified.	
			Hex 21 = Terminate Thread instruction issued against the initial thread by a thread in the process.	
			Hex 20 = Terminate Process instruction issued by a thread within the process.	
			Hex 18 = An unhandled signal with a default signal handling action of *terminate the process* or *terminate the request* was delivered to the process.	
			Hex 10 = Exception was not handled by the initial thread in the process.	
			Hex 00 = Process terminated externally.	
12	C		Initial internal termination code	Bits 8-23
			The code is assigned in one of the following ways: 1. If the termination is caused by a Return External instruction from the first invocation in the initial thread, then this code is binary 0's. 2. If the termination is caused by an unhandled signal, then this code is the signal number of the unhandled signal. 3. The code is assigned by the original exception code that caused process termination to start.	
			Note: The *process initial internal termination status* represents the final internal termination status prior to entering the termination phase. It is updated by the most recent internal termination action. It is possible for both the *process initial internal termination status* and the *process initial external termination status* fields to contain valid non-zero values.	
15	F	Process initial external termination status	Char(3)	
15	F		Initial external termination reason:	Bits 0-7

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			Hex 80 = Terminate Process instruction issued explicitly to the process by a thread in another process.	
			Hex 40 = Terminate Thread instruction issued explicitly to the initial thread of the process by a thread in another process.	
			Hex 00 = Process terminated internally.	
15	F		Initial external termination code:	Bits 8-23
			Note: The *process external internal termination status* represents the final external termination status prior to entering the termination phase. It is updated by the most recent external termination action. It is possible for both the *process initial internal termination status* and the *process initial external termination status* fields to contain valid non-zero values.	
18	12	Process final termination status	Char(3)	
18	12		Final termination reason:	Bits 0-7
			Hex 80 = Return instruction from first invocation.	
			Hex 41 = Terminate Thread instruction issued against the initial thread in the process by a thread within this process.	
			Hex 40 = Terminate Process instruction issued by a thread within the process.	
			Hex 21 = Terminate Thread instruction issued against the initial thread in the process by a thread in another process.	
			Hex 20 = Terminate Process instruction issued for the process by a thread in another process.	
			Hex 18 = An unhandled signal with a default signal handling action of *terminate the process* or *terminate the request* was delivered to the process.	
			Hex 10 = Exception was not handled by the initial thread in the process.	

Offset Dec	Hex	Field Name	Data Type and Length	
18	12		Final termination code Assigned in one of the following ways: 1. If the termination is caused by a Return External instruction from the first invocation in the initial thread, then this code is binary 0's. 2. If the termination is caused by an unhandled signal, then this code is the signal number of the unhandled signal. 3. The code is assigned by the original exception code that caused process termination to start. The process final termination status is presented as event-related data in the terminate process event. Usually the event is the only source of the process final termination status since the process will cease to exist before its attributes can be materialized. **Note:** The *process final termination status* describes how the final phase of the process terminated. It is updated by the most recent termination action for the final process phase.	Bits 8-23
21	15	—- End —-		

- Hex 21 = Process resource usage attributes

  Process resource usage attributes are materialized when the value of operand 3 is hex 21. The format and associated values of this attribute are as follows:

Offset Dec	Hex	Field Name	Data Type and Length
8	8	Temporary auxiliary storage used in bytes	Bin(4)
12	C	Total processor time used	Char(8)
20	14	Number of process scoped locks currently held by the process (including implicit locks)	Bin(2)
22	16	Reserved (binary 0)	Char(2)
24	18	Temporary auxiliary storage used in megabytes	UBin(4)
28	1C	Reserved (binary 0)	UBin(4)
32	20	Database processor time used	Char(8)
40	28	Page faults	Char(8)
48	30	Synchronous database reads	Char(8)
56	38	Asynchronous database reads	Char(8)
64	40	Synchronous database writes	Char(8)
72	48	Asynchronous database writes	Char(8)
80	50	Synchronous non-database reads	Char(8)
88	58	Asynchronous non-database reads	Char(8)
96	60	Synchronous non-database writes	Char(8)
104	68	Asynchronous non-database writes	Char(8)
112	70	—- End —-	

A process which has used more than 2,147,483,647 bytes of temporary auxiliary storage will have 2,147,483,647 returned for *temporary auxiliary storage used in bytes*. *Temporary auxiliary storage used in*

megabytes will be the amount of temporary auxiliary storage rounded down to the nearest megabyte (i.e. if a process was using 900 KB, 0 would be returned).

The **database processor time used** is the total amount of processor time used performing database processing. If the system does not support this metric, a value of hex 0000000000000000 is returned. If the system does support this and needs to return a value of 0, a value of hex 0000000000001000 is returned. For all other cases, the significance of bits within this field is the same as that defined for the time-of-day clock. See "Standard Time Format" on page 1272 for additional information.

The **page faults** field is the number of page faults incurred.

The **synchronous database reads** field is the number of synchronous reads into main storage done while performing database processing.

The **asynchronous database reads** field is the number of asynchronous reads into main storage done while performing database processing.

The **synchronous database writes** field is the number of synchronous writes from main storage done while performing database processing.

The **asynchronous database writes** field is the number of asynchronous writes from main storage done while performing database processing.

The **synchronous non-database reads** field is the number of synchronous reads into main storage done while performing non-database processing.

The **asynchronous non-database reads** field is the number of asynchronous reads into main storage done while performing non-database processing.

The **synchronous non-database writes** field is the number of synchronous writes from main storage done while performing non-database processing.

The **asynchronous non-database writes** field is the number of asynchronous writes from main storage done while performing non-database processing.

- Hex 22 = Obsolete

  A materialization option value of hex 22 causes this information to be returned:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	—- End —-		

  This obsolete information is not supplied with materialization option hex 00.

- Hex 23 = Thread performance attributes

  Thread performance attributes are materialized when the value of operand 3 is hex 23. The format and associated values of this attribute are as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
8	8	Number of synchronous page reads into main storage associated with data base	Bin(4)
12	C	Number of synchronous page reads into main storage not associated with data base	Bin(4)
16	10	Total number of synchronous page writes from main storage This includes writes associated with and not associated with data base.	Bin(4)
20	14	Number of transitions into ineligible wait state	UBin(2)
22	16	Number of transitions into an instruction wait state	UBin(2)
24	18	Number of transitions into ineligible wait state from an instruction wait	UBin(2)

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
26	1A	Timestamp of materialization (local time)	Char(8)
34	22	Number of asynchronous reads into main storage associated with data base	Bin(4)
38	26	Number of asynchronous reads into main storage not associated with data base	Bin(4)
42	2A	Number of synchronous writes from main storage associated with data base	Bin(4)
46	2E	Number of synchronous writes from main storage not associated with data base	Bin(4)
50	32	Number of asynchronous writes from main storage associated with data base	Bin(4)
54	36	Number of asynchronous writes from main storage not associated with data base	Bin(4)
58	3A	Total number of writes from main storage of permanent objects	Bin(4)
62	3E	Reserved (binary 0)	Bin(4)
66	42	Number of page faults on process access group objects	Bin(4)
70	46	Number of internal effective address overflow exceptions	Bin(4)
74	4A	Number of internal binary overflow exceptions	Bin(4)
78	4E	Number of internal decimal overflow exceptions	Bin(4)
82	52	Number of internal floating point overflow exceptions	Bin(4)
86	56	Number of times a page fault occurred on an address that was currently part of an auxiliary storage I/O operation	Bin(4)
90	5A	Number of times the process explicitly waited for outstanding asynchronous I/O operations to complete	Bin(4)
94	5E	Number of page faults for machine index objects	Bin(4)
98	62	Thread processor time used	Char(8)
106	6A	—- End —-	

Each of the UBin(2) counters has a limit of 65,535. If this limit is exceeded, the count is set to 0, and no exception is signaled.

The *thread performance attributes* are not supplied with materialization option hex 00.

- Hex 24 = Execution status attributes

  Execution status attributes are materialized when the value of operand 3 is hex 24. The format and associated values of this attribute are as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
8	8	Priority	Char(2)	
8	8		Process priority	Char(1)
9	9		Thread priority adjustment	Char(1)
10	A	Pending thread interrupts	Char(2)	
10	A		Time slice end	Bit 0
10	A		Transfer lock	Bit 1
10	A		Asynchronous lock retry	Bit 2
10	A		Suspend process	Bit 3
10	A		Resume process	Bit 4
10	A		Modify resource management attribute	Bit 5
10	A		Modify process or thread attribute	Bit 6
10	A		Terminate machine processing	Bit 7
10	A		Terminate process or thread	Bit 8

Offset Dec	Offset Hex	Field Name	Data Type and Length	
10	A		Wait time-out	Bit 9
10	A		Event schedule	Bit 10
10	A		Thread operations between threads	Bit 11
10	A		Cancel long running instruction	Bit 12
10	A		Reserved (binary 0)	Bit 13
10	A		Deliver queue space message	Bit 14
10	A		Signal schedule	Bit 15
12	C	Thread execution status	Char(2)	
12	C		Suspended by Suspend Process	Bit 0
12	C		Instruction wait	Bit 1
12	C		In MPL	Bit 2
12	C		Ineligible wait	Bit 3
12	C		In kernel mode	Bit 4
12	C		Stopped by a signal	Bit 5
12	C		Suspended by Suspend Thread	Bit 6
12	C		Reserved (binary 0)	Bits 7-15
14	E	Thread wait status	Char(2)	
14	E		Wait on event	Bit 0
14	E		Dequeue	Bit 1
14	E		Lock	Bit 2
14	E		Wait on time	Bit 3
14	E		Wait to start a commit cycle	Bit 4
14	E		Wait on mutex (includes both mutex types)	Bit 5
14	E		Wait on select function	Bit 6
14	E		Wait on signal	Bit 7
14	E		Wait on unowned resource	Bit 8
14	E		Wait on thread	Bit 9
14	E		Wait on transaction control structure	Bit 10
14	E		Wait on condition	Bit 11
14	E		Wait on semaphore	Bit 12
14	E		Wait on Java(TM)	Bit 13
14	E		Reserved (binary 0)	Bits 14-15
16	10	Process class identification	Char(2)	
16	10		Main storage pool ID	Char(1)
17	11		Multi-programming level class ID	Char(1)
18	12	Process processor time used	Char(8)	
26	1A	Thread performance attributes	Char(82)	
26	1A		Number of synchronous reads into main storage associated with data base	Bin(4)
30	1E		Number of synchronous reads into main storage not associated with data base	Bin(4)
34	22		Total number of synchronous page writes from main storage This includes writes associated with and not associated with data base.	Bin(4)
38	26		Transitions to ineligible wait	UBin(2)
40	28		Transitions to instruction wait	UBin(2)
42	2A		Transitions to ineligible from instruction wait	UBin(2)
44	2C		Number of asynchronous reads into main storage associated with data base	Bin(4)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
48	30		Number of asynchronous reads into main storage not associated with data base	Bin(4)
52	34		Number of synchronous writes from main storage associated with data base	Bin(4)
56	38		Number of synchronous writes from main storage not associated with data base	Bin(4)
60	3C		Number of asynchronous writes from main storage associated with data base	Bin(4)
64	40		Number of asynchronous writes from main storage not associated with data base	Bin(4)
68	44		Total number of writes from main storage of permanent objects	Bin(4)
72	48		Reserved (binary 0)	Bin(4)
76	4C		Number of page faults on process access group objects	Bin(4)
80	50		Number of internal effective address overflow exceptions	Bin(4)
84	54		Number of internal binary overflow exceptions	Bin(4)
88	58		Number of internal decimal overflow exceptions	Bin(4)
92	5C		Number of internal floating point overflow exceptions	Bin(4)
96	60		Number of times a page fault occurred on an address that was currently part of an auxiliary storage I/O operation	Bin(4)
100	64		Number of times the process explicitly waited for outstanding asynchronous I/O operations to complete	Bin(4)
104	68		Number of page faults for machine index objects	Bin(4)
108	6C	Active threads	UBin(4)	
112	70	—- End —-		

The **wait on unowned resource** thread wait status field indicates that the materialized thread is waiting on one of several general purpose resources used internally by the system.

The **active threads** field contains a count of the current number of active threads in the process at the time of the materialization. An *active thread* may be either actively running, suspended or waiting on a resource.

- Hex 25 = Process control space pointer

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Process control space pointer	System pointer
32	20	—- End —-	

A system pointer to the *process control space* is materialized when the value of operand 3 is hex 25. If a process control space pointer is supplied in operand 2, it is ignored. A pointer to the process that is executing the MATPRATR instruction is always materialized.

- Hex 26 = Group profile list

  A materialization option value of hex 26 causes the *group profile list* to be materialized as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Pointer to the group profile list (original, provided by MI user)	Space pointer

Offset Dec	Hex	Field Name	Data Type and Length
32	20	Number of user profiles in the encapsulated group profile list	Bin(2)
34	22	Reserved (binary 0)	Char(14)
48	30	List of user profiles in the encapsulated group profile list (one system pointer to each user profile in the list)	[*] System pointers
*	*	—- End —-	

The *pointer to the group profile list* is the space pointer provided by the MI user. No verification of this pointer or the contents of the group profile list pointed to by this space pointer is done.

The *group profile list* is not supplied with materialization option hex 00.

- Hex 27 = Group profile list option

  A materialization option's value of hex 27 causes the *process control attributes* to be materialized. The format of the *process control attributes* materialization is defined in prior text for this instruction.

- Hex 28 = Process category

Offset Dec	Hex	Field Name	Data Type and Length
8	8	Process category	Char(2)
10	A	—- End —-	

- Hex 29 = Queue space object pointer

Offset Dec	Hex	Field Name	Data Type and Length
16	10	Queue space object pointer	System pointer
32	20	—- End —-	

The system pointer with addressability to the queue space is placed in the space addressed by operand 1.

- Hex 2A = Secondary process communications object (PPCO) pointer

Offset Dec	Hex	Field Name	Data Type and Length
16	10	PPCO pointer	Space pointer
32	20	—- End —-	

The PPCO space pointer is placed in the space addressed by operand 1. If this option is specified by a user state program, an *object domain or hardware storage protection violation* (hex 4401) exception will be signaled.

- Hex 2B = Signal enablement option

  A materialization option's value of hex 2B causes the *process control attributes* to be materialized. The format of the *process control attributes* materialization is defined in prior text for this instruction.

- Hex 2C = Process signal controls

  Process signal controls are materialized when the value of operand 3 is hex 2C. The format and associated values of this attribute are as follows:

Offset Dec	Hex	Field Name	Data Type and Length	
8	8	Pending signals mask	Char(8)	
8	8		Reserved (binary 0)	Bit 0
8	8		Pending signal status	Bits 1-63

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = No signal is present	
			1 = A signal has been received, the signal action is blocked	
16	10	Signal blocking mask	Char(8)	
16	10		Reserved (binary 0)	Bit 0
16	10		Blocked/unblocked option	Bits 1-63
			0 = Signal is unblocked. Signal action for the signal monitor is eligible to be scheduled.	
			1 = Signal is blocked. Signal action for the signal monitor is to be deferred.	
24	18	Number of signal monitors	Bin(4)	
28	1C	Reserved (binary 0)	Char(4)	
32	20	Signal monitor data	[*] Char(16)	
		(repeated for each signal monitor)		
32	20		Signal number	Bin(4)
36	24		Signal action	Bin(2)
			-1 = Signal associated with this signal monitor is not supported	
			0 = Handle using signal default action	
			1 = Ignore the signal (discard)	
			2 = Handle the signal by executing signal catching function	
38	26		Signal default action	Bin(2)
			0 = Terminate the process	
			1 = Terminate the request	
			2 = Ignore the signal (discard)	
			3 = Stop the process	
			4 = Continue the process if stopped	
			5 = Signal exception	
40	28		Maximum number of signals to be retained	Bin(2)
42	2A		Current number of pending signals	Bin(2)
44	2C		Signal priority (1-255; 1 = highest priority)	Bin(2)
46	2E		Reserved (binary 0)	Char(2)
*	*	—- End —-		

The **pending signals mask** is used to determine if signals have been received for signal monitors whose signal handling actions have been blocked from delivery. The **pending signal status** specified for the *nth* bit position in *pending signals mask* is applied to the *nth* signal monitor defined for the process. If *pending signal status* is binary 1, a signal for the corresponding signal monitor has been received by the process and the signal action for the monitor is blocked from delivery in all threads within the process. If *pending signal status* is binary 0, there are no pending signals at the process level for the corresponding signal monitor.

The **signal blocking mask** field is used to determine if the signal action for the associated signal monitor is eligible to be scheduled. The *blocked/unblocked option* specified for the *nth* bit position in the *signal blocking mask* is applied to the *nth* signal monitor in the *signal monitors attributes*. When the signal is **unblocked**, the signal action for the signal monitor associated with the signal is eligible to be

scheduled. When the signal is **blocked**, the signal will be retained, up to the limit set by the *maximum number of signals to be retained* value in the signal monitor associated with the signal. The *signal blocking mask* is a thread resource, the value materialized is obtained from the thread being materialized.

The **number of signal monitors** indicates the actual number of signal monitor data entries returned in the materialization template. Partial signal monitor data entries are not returned. The machine supports a maximum of 63 signal monitors. The *number of signal monitors* is a process resource, the value materialized is obtained from the process being materialized.

The **signal monitor data** defines the attributes for each signal supported. The order in which the signal monitors are defined will determine the signal monitor used by the machine for the generation and delivery of a signal. The *signal monitor data* is a process resource, the values materialized are obtained from the process being materialized.

The **signal number** is defined by the MI user and has no significance to the machine.

The **signal action** defines the action to be taken by the machine upon receipt of the signal by the process. If the value of *signal action* is *signal associated with this signal monitor is not supported*, the *signal default action* and *maximum number of signals to be retained* fields for this signal monitor are ignored by the machine and will be set to binary 0.

The **signal default action** field defines the action to be taken by the machine when the *signal action* is set to *handle using signal default action*. The *terminate the process* action will place the process in termination phase, allowing invocation exits to be invoked. If the process is already in termination phase, the *terminate the process* action is ignored. The *terminate the request* action will result in the cancellation of all invocations up to the nearest invocation that has an invocation status of request processor. If an invocation with an invocation status of request processor is not present, the *terminate the process* action is taken. The *stop the process* action will result in the execution of the process being temporarily suspended until a signal is generated for the process that has *continue the process if stopped* as its *signal default action*. When a process is in the *stopped* state, the normal process control functions remain in effect (the process may be suspended, resumed or terminated). The *signal exception* action will result in the *asynchronous signal received* (hex 4C03) exception being signaled by the machine, with the signal monitor number and signal-specific data being included in the exception-related data.

The **maximum number of signals to be retained** field indicates the number of signals that the machine retains when the signal action associated with the signal monitor can not be taken (held pending). The *maximum number of signals to be retained* limits the number of signals held pending for the signal monitor for the process and for each thread.

The **current number of pending signals** is the number of signals held pending at the process level for this signal monitor whose signal handling action has not been scheduled.

The **signal priority** specifies the relative importance of this signal compared with other signals being monitored within a process. The *signal priority* establishes the order in which signal handling actions are scheduled if multiple signal monitors have been signaled. Signal handling actions will be taken in the order the signal monitors were signaled when multiple signal monitors have the same priority.

- Hex 2D = Lock statistics

Offset Dec	Offset Hex	Field Name	Data Type and Length
8	8	Number of data base lock waits	Bin(4)
12	C	Number of non-data base lock waits	Bin(4)
16	10	Number of internal machine lock waits	Bin(4)
20	14	Time spent on data base lock waits	Char(8)
28	1C	Time spent on non-data base lock waits	Char(8)
36	24	Time spent on internal machine lock waits	Char(8)
44	2C	—- End —-	

The three lock counters, **number of data base lock waits**, **number of non-data base lock waits**, and **number of internal machine lock waits**, are the number of times that the thread has had to wait to obtain a lock of the specified type.

The three time fields, **time spent on data base lock waits**, **time spent on non-data base lock waits**, and **time spent on internal machine lock waits**, are the cumulative amount of time that the thread had to wait for the locks of the specified type. The time is of the same format as the time of day clock. See "Standard Time Format" on page 1272 for additional information on the format of the time of day clock.

- Hex 2E = Threads enablement option

  A materialization option's value of hex 2E causes the *process control attributes* to be materialized. The format of the *process control attributes* materialization is defined in prior text for this instruction.

- Hex 2F = Secondary threads control option

  A materialization option's value of hex 2F causes the *process control attributes* to be materialized. The format of the *process control attributes* materialization is defined in prior text for this instruction.

- Hex 30 = Initial thread information

  The thread identifier and thread handle for the initial thread of the identified process are materialized when the value of operand 3 is hex 30. The format and associated values of this attribute are as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
8	8	Active threads	UBin(4)	
12	C	Reserved (binary 0)	Char(4)	
16	10	Thread identifier	Char(8)	
24	18	Thread handle	UBin(4)	
28	1C	Thread attributes	Char(1)	
28	1C		Thread type	Bit 0
			0 = User thread	
28	1C		Reserved (binary 0)	Bits 1-7
29	1D	Reserved (binary 0)	Char(3)	
32	20	Resources affinity identifier	UBin(4)	
36	24	Reserved (binary 0)	Char(12)	
48	30	—- End —-		

The **active threads** field contains a count of the current number of active threads in the process at the time of the materialization. An *active thread* may be either actively running, suspended or waiting on a resource. Subsequent materializations may result in a different *active threads* value.

The **thread identifier** is an identifier for the initial thread which is unique within the process being materialized. While no two threads initiated within the same process will have the same identifier, it is possible that threads in different processes may have the same value for the identifier.

The **thread handle** is an identifier for the initial thread which is unique within the process being materialized. The *thread handle* returned is an unopened handle and may be used on other thread management instructions to uniquely identify a thread within the process being materialized. Thread handles are not unique system wide. Thus, threads from different processes may have the same value for their thread handle.

The **thread type** field determines how the thread was initiated. The initial thread of a process is always a *user thread*.

The **resources affinity identifier** is the initial thread's value for *resources affinity identifier*.

Warning: Temporary Level 3 Header

Authorization Required

•

• Process control special authorization

  –

  – For materializing a process other than the one containing the thread issuing this instruction

• Execute

  –

  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize

  –

  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

0A04 Special Authorization Required

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

28 Process/Thread State

2802 Process Control Space Not Associated with a Process

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Process Locks (MATPRLK)

Op Code (Hex)	Operand 1	Operand 2
0312	Receiver	Process control space

Operand 1: Space pointer.

Operand 2: System pointer or null.

Bound program access

```
Built-in number for MATPRLK is 51.
MATPRLK (
         receiver               : address
         process_control_space  : address of system pointer OR
                                  null operand
)
```

Description: The lock status of the process identified by operand 2 is materialized into the *receiver* specified by operand 1. If operand 2 is null, the lock status is materialized for the process containing the thread issuing the instruction. The lock status is materialized for each lock allocated to the process and for each lock allocated to each thread contained in the process. The materialization identifies each object, object location, or space location for which the process or applicable thread has a lock allocated or for which the applicable thread is in a synchronous or asynchronous wait. The format of the materialization is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Materialization size specification	Char(8)
0	0		Number of bytes provided for materialization
4	4		Number of bytes available for materialization
8	8	Number of lock entries	Bin(2)
10	A	Expanded number of lock entries	Bin(4)
14	E	Reserved (binary 0)	Char(2)
16	10	Lock status	[*] Char(32)
		(repeated for the number of lock entries)	
16	10		Object, object location, space location or binary 0 (if no pointer exists)
32	20		Lock state
32	20		LSRD
32	20		LSRO
32	20		LSUP
32	20		LEAR
32	20		LENR
32	20		Reserved (binary 0)
33	21		Status of lock state for process
33	21		Lock scope object type **0 =** Process control space **1 =** Transaction control structure
33	21		Lock scope **0 =** Lock is scoped to the *lock scope object type* **1 =** Lock is scoped to a thread in the process
33	21		Object, object location, or space location no longer exists
33	21		Waiting because this lock is not available

Offset Dec	Hex	Field Name	Data Type and Length	
33	21		Thread is in asynchronous wait for lock	
33	21		Thread is in synchronous wait for lock	
33	21		Implicit lock (machine-applied)	
33	21		Lock held by a process, thread, or transaction control structure	
34	22		Lock information	Ch
34	22		Reserved	
34	22		Lock is held by a process, thread or transaction control structure other than the current process or thread	
34	22		Lock is held by the machine	
35	23		Reserved (binary 0)	Ch
36	24		Unopened thread handle	UB
40	28		Thread ID	Ch
*	*	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception described previously) are signaled if the *receiver* contains insufficient area for the materialization.

The **number of lock entries** field identifies the number of lock entries that are materialized. When a process and its threads hold more than 32,767 locks, this field is set with its maximum value of 32,767. This field has been retained in the template for compatibility with programs using the template prior to the changes made to support materialization of more than 32,767 lock entries.

The **expanded number of lock entries** field identifies the number of lock entries that are materialized. This field is always set in addition to the number of lock entries field described previously; however, it does not have a maximum limit of 32,767, so it can be used to specify that more than 32,767 locks have been materialized. When a process and its threads hold more than 32,767 locks, the *number of lock entries* field will equal 32,767, which would be incorrect. The *expanded number of lock entries* field, however, will identify the correct number of lock entries materialized. In all cases, this field should be used instead of the *number of lock entries* field to get the correct count of lock entries materialized.

The **lock scope** field identifies the scope of the lock being requested. If the lock is allocated, the **lock holder information** field is binary 0. If the lock is pending, the *lock information* field contains information about the current holder of the lock.

For allocated locks that are process scope, the **unopened thread handle** and **thread ID** fields will be set to binary 0. For allocated locks that are thread scope, these fields will identify the specific thread in the specified process that holds the lock. For locks being waited on, these fields will identify the specific thread in the specified process that is waiting for the lock, regardless of the *lock scope* value.

For space location locks which have been acquired using the LOCKTSL instruction with the *type of teraspace storage location lock* field set to binary 1, a null pointer value is returned. A null pointer value is also returned for any lock on a teraspace storage location which is not held by a thread in the current process.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

  –

  – Context referenced by address resolution

Lock Enforcement

•

• Materialize

  –

  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

28 Process/Thread State

2802 Process Control Space Not Associated with a Process

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Process Message (MATPRMSG)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
039C	Materialization template	Message template	Source template	Selection template

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Space pointer or null.

Operand 4: Space pointer.

Bound program access

```
Built-in number for MATPRMSG is 127.
MATPRMSG (
          receiver_template    : address
          message_template     : address
          source_template      : address OR
                                 null operand
          selection_template   : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note

It is recommended that you use selection types 8 and 9 for 8-byte invocation and activation group marks, respectively, rather than selection types 3 and 4. It is also recommended that you use the 8-byte invocation and activation group marks at the end of the materialization template. 4-byte marks can wrap and produce unexpected results.

Description: A message is materialized from a queue space according to the options specified. The message is located on a queue space queue specified by operand 3. The message is selected by the operand 4 criteria. Operands 1 and 2 contain the materialized information from the process message.

The template identified by operand 1 must be 16-byte aligned. Following is the format of the materialization template:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Template size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Process queue space queue offset	Bin(4)	
12	C	Reserved (binary 0)	Char(4)	
16	10	Time sent (local time)	Char(8)	
24	18	Time modified (local time)	Char(8)	
32	20	Interrupted invocation	Invocation pointer	
48	30	Target invocation	Invocation pointer or System pointer	
64	40	Original target invocation	Invocation pointer	
80	50	Source program location	Suspend pointer	
96	60	Target program location	Suspend pointer	
112	70	Originating program location	Suspend pointer	
128	80	Invocation mark	UBin(4)	
132	84	Activation group mark	UBin(4)	
136	88	Thread ID	Char(8)	
144	90	Invocation mark	UBin(8)	
		For Non-Bound programs, the following datatype should be used:		
144	90		Invocation mark (Non-Bound program)	Char(8)
152	98	Activation group mark	UBin(8)	
		For Non-Bound programs, the following datatype should be used:		
152	98		Activation group mark (Non-Bound program)	Char(8)
160	A0	—- End —-		

The first 4 bytes of the materialization template identify the total **number of bytes provided** for use by the instruction. This number is supplied as input to the instruction and is not modified by the instruction. If the value is zero for this field, then the operand 1 template is not returned. A number less than 128 (but not 0) causes the *materialization length invalid* (hex 3803) exception.

Process queue space queue offset	This value indicates which queue in the queue space a message resides on. A value of -1 will be returned if the message is on the external queue, and zero if the message is on the message log. If the message resides on an invocation queue this field will be zero.
Time sent (local time)	The value of the system time-of-day clock when the message was originally sent (Signal Exception (SIGEXCP) or message originated as a result of an exception). See "Standard Time Format" on page 1272 for additional information on the time-of-day clock.
Time modified (local time)	This value is initially equal to the *time sent* value. See "Standard Time Format" on page 1272 for additional information on the time-of-day clock.
Interrupted invocation	An invocation pointer that addresses the invocation which currently has this message as its interrupt cause. This pointer will have a null pointer value if the message is not an exception message.
Target invocation	An invocation pointer or system pointer that identifies which queue contains the message. If the message resides on an invocation queue, this field contains an invocation pointer that addresses the invocation whose invocation message queue currently contains the message. If the message does not reside on an invocation queue, then a system pointer to the Queue Space is returned. The *process queue space queue offset* field indicates on which queue the message resides, the message log or the external queue.
Original target invocation	An invocation pointer that addresses the invocation which originally was sent the message. This pointer will have a null pointer value if the original target invocation no longer exists.
Source program location	A suspend pointer which identifies the program, module, procedure, and statement where the source invocation was suspended (due to a CALL or some form of interrupt).
Target program location	A suspend pointer which identifies the program, module, procedure, and statement where the target invocation was suspended (due to a CALL or some form of interrupt). If the message is no longer in an invocation message queue, then this pointer reflects the invocation of the last (most recent) invocation message queue in which the message resided. This pointer will have a null pointer value if the message has never resided in an invocation message queue.
Originating program location	A suspend pointer which identifies the program, module, procedure, and statement of the instruction that sent the message. For messages sent by the machine, this pointer area contains a machine-dependent representation of the source machine component.
Invocation mark	If the message has ever resided on an invocation, this field will be non-zero. It contains the mark of the invocation where the message was last queued. The value returned in the 4-byte invocation mark may have wrapped.
Activation group mark	If the message has ever resided on an invocation, this field will be non-zero. It contains the mark of the activation group which contains the invocation where the message was last queued. The value returned in the 4-byte activation group mark may have wrapped.

Thread ID

This is the thread ID of the thread that added the message to the process queue space.

Messages will contain a thread ID that is determined by the queue space to which the message is being sent. When sending to the current process queue space, the thread ID of the current thread will be used. When sending to a different process queue space, the thread ID of the initial thread for the owning process will be used. If the queue space is not associated with a process, the thread ID will be 0.

Note: When sending a reply, the message will contain the thread ID from the associated inquiry message.

The template identified by operand 2 is used to contain the materialized message. It must be 16-byte aligned with the following format.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Message type	Char(1)	

Hex 00 = Informational message 0

Hex 01 = Informational message 1

Hex 04 = Exception message

Hex 06 = Return/Transfer Control message

Hex 07 = Return message

Hex 10 = Control Entry message

Hex 11 = Escape/return handler

All other values are reserved.

9		
9		
Reserved (binary 0)		
Char(1)		
10		
A		
Message severity		
Bin(2)		
12		
C		
Reply/Inquiry message reference key		
Char(4)		
16		
10		
Message status mask		
Char(8)		
16		
10		
Log message		
Bit 0		

0 = The message is not queued to the Process Message Log.

1 = The message is queued to the Process Message Log.

16
10
Inquiry
Bit 1

0 = Message will not accept a reply.

1 = Message will accept a reply.

16
10
Reply
Bit 2

0 = Message does not represent a *reply* message.

1 = Message represents a *reply* message.

16
10
Answered
Bit 3

0 = For messages with a *status* of *inquiry*, this indicates that a reply message has not been received.

1 = For messages with a *status* of *inquiry*, this indicates that a reply message has been received.

16
10
Message being processed
Bit 4

0 = This value of the flag has no particular meaning.

1 = For a *message type* of *exception*, this indicates that the interrupt is currently being handled.

16
10
Retain
Bit 5

0 = Message will be dequeued and its contents discarded when the following message status bits are zero: *log*, *message being processed*, and *action pending*.

1 = Keeps the message from being dequeued after all other message status bits are zero.

16
10
Action pending
Bit 6

0 = Indicates no actions are pending based on this message.

1 = Indicates that the message is either an interrupt cause or is a *return, return/transfer control message.*

16
10
Invoke Process Default Exception Handler (PDEH)
Bit 7

0 = Do not invoke PDEH.

1 = Invoke PDEH, only valid if *message type* is *exception*.

16
10
Error
Bit 8

0 = No error has occurred in the sending of this message.

1 = An error has occurred in the sending of this message.

16
10
PDEH previously invoked
Bit 9

0 = PDEH has not previously been invoked for this message.

1 = PDEH has previously been invoked for this message.

16
10
Reserved (binary 0)
Bits 10-31
16
10
User defined status
Bits 32-63
24
18
Interrupt class mask
Char(8)

Note: All fields in the interrupt class mask that are marked as 'Reserved' have a value of binary zero.

24
18
Binary overflow or divide by zero
Bit 0
24
18
Decimal overflow or divide by zero
Bit 1
24
18
Decimal data error
Bit 2
24
18
Floating-point overflow or divide by zero
Bit 3
24
18
Floating-point underflow or inexact result
Bit 4
24
18
Floating-point invalid operand or conversion error
Bit 5
24
18
Other data error (edit mask, etc)
Bit 6
24
18
Specification (operand alignment) error
Bit 7
24
18
Pointer not set/pointer type invalid
Bit 8
24
18
Object not found
Bit 9
24
18
Object destroyed
Bit 10
24

18
Address computation underflow/overflow
Bit 11
24
18
Space not allocated as specified offset
Bit 12
24
18
Domain/State protection violation
Bit 13
24
18
Authorization violation
Bit 14
24
18
Java thrown class
Bit 15
24
18
Reserved
Bits 16-28
24
18
Other MI generated exception (not function check)
Bit 29
24
18
MI generated function check/machine check
Bit 30
24
18
Message generated by Signal Exception instruction
Bit 31
24
18
Reserved
Bits 32-39
24
18
User defined
Bits 40-63
32
20
Initial handler priority
Char(1)
33
21
Current handler priority
Char(1)
34
22
Exception ID
UBin(2)
36
24
PDEH reason code
Char(1)
37
25
Signal class
Char(1)
38
26
Compare data length
Bin(2)
40
28

Message ID
Char(7)
47
2F
Reserved (binary 0)
Char(1)
48
30
Max message data length
Bin(4)
Input to the instruction
52
34
Message data length
Bin(4)
56
38
Max length of message extension data
Bin(4)
Input to the instruction
60
3C
Message extension data length
Bin(4)
64
40
Message data pointer
Space pointer
Input to the instruction
80
50
Message data extension pointer
Space pointer
Input to the instruction
96
60
Message format information
Char(32)
If the message is not a *return* or *return/transfer control* message.
96
60
Compare data
Char(32)
If the message is a *return* or *return/transfer control* message.
96
60
Return handler identifier
System pointer or Procedure pointer
112
70
Reserved
Char(16)
128
80
Reserved (binary 0)
Char(48)
176
B0
—- End —-

The first 4 bytes of the message template identify the total **number of bytes provided** for use by the instruction. This number is supplied as input to the instruction and is not modified by the instruction. A number less than 160 causes the *materialization length invalid* (hex 3803) exception.

Message type

This value determines the type of the message. The type of message determines which *message status values* have meaning.

The following *message status attributes* are valid for all *informational message types:*.

- Log Message
- Reply
- Inquiry

The following *message status attributes* are valid for a *message type* of *exception:*.

- Log Message
- Retain
- Action pending
- Invoke PDEH
- Inquiry
- Reply

The following *message status attributes* are valid for a *message type* of *return* or *return/transfer control*.

- Action pending

The following describes each *message type* in detail.

-
- *Informational 0* - There are no special requirements as to what message status contains or what the Target Invocation pointer actually identifies.
- *Informational 1* - There are no special requirements as to what message status contains or what the Target Invocation pointer actually identifies.
- *Exception* - This type of message has an *interrupt cause* for the *interrupted invocation*.
- *Return/transfer control* - This message type indicates that a return handler is invoked when the target invocation is exited for any reason including XCTL. Additionally, messages of this type interpret the *message format information* field to identify a program or procedure to be invoked as the return handler.
- *Return* - This message type indicates that a return handler will be invoked if the target invocation is exited for any reason other than Transfer Control (XCTL). In the case of XCTL, the message is preserved and associated with the transferred to invocation. Additionally, messages of this type interpret the *message format information* field to identify a program or procedure to be invoked as the return handler.
- *Control Entry* - This message type is associated with an invocation. The *control entry* is used to keep data required for exception processing.
- *Escape/Return Handler* - This message type is associated with an invocation. The *escape/return handler* is used to keep data required for processing when control is returned to an invocation.

Message severity

A value indicating the severity of the message.

Reply/Inquiry message reference key

If the message materialized is an *inquiry* message that has been answered, this is the *message reference key* of its reply message.

If the message materialized is a *reply* message, this is the *message reference key* of its inquiry message.

This value only has meaning for *exception* and *informational* messages.

Message status mask

A bit-significant value indicating the original status of the message.

-
- **Log message** status. If this bit is 1, the message is queued to the Process Message Log until it is explicitly removed.
- **Inquiry** status. If this bit is 1, this message will accept a *reply* message.
- **Reply** status. If this value is 1, this message is a reply to an *inquiry* message. The *reply message reference key* is used to identify the message for which the message was replied.
- **Answered** status. If this value is 1, the message is a inquiry message for which a reply has been sent. The *reply message reference key* is used to identify the reply message.
- **Retain** status. If this bit is 1, the message is kept even if the invocation message queue after the following message status bits are zero: *log*, *message being processed*, and *action pending*.
- **Action pending** status. If this bit is 1, this message represents an exception which is the current interrupt cause for the specified Source Invocation or else it is a *return* or *return/transfer control* message which has not yet been processed.
- **Invoke Process Default Exception Handler**. This status only has meaning for *exception* messages.

Interrupt class mask

A bit-significant value indicating the cause of the interrupt. The MI user is allowed to use the machine-defined classes since machine-generated errors may be re-sent by the MI user.

This value only has meaning for *exception* messages.

The **Java thrown class** interrupt class indicates that the message corresponds to a thrown class in the Java language.

Initial handler priority

An unsigned eight-bit binary number which selects the initial interrupt handler priority. This value is within the range of 64 - 255.

This value only has meaning for *exception* messages.

Exception ID

A two-byte field that identifies the exception being defined by this message.

This value only has meaning for *exception* messages.

PDEH reason code

A value defined by the user which indicates the type of processing to be attempted by the Process Default Exception Handler.

This value only has meaning for *exception* messages.

Signal class

A value defined by the user which is used to select exception monitors.

This value only has meaning for *exception* messages.

Compare data length

A value indicating the number of bytes provided as compare data.

This value only has meaning for *exception* messages.

Message ID

Specifies the message identifier of a message description whose predefined message is being sent.

Max message data length

Input to the instruction that specifies the number of bytes supplied for the message data. The maximum value allowed is 65,504.

Message data length

A value indicating the number of bytes of message data for this process message.

Max extension data length

Input to the instruction that specifies the number of bytes supplied for the message data extension. The maximum value allowed is 65,504.

Message extension data length

A value indicating the number of bytes of message data extension for this process message.

Message data pointer

A pointer to the area to receive the message data. This field is ignored if the *max message data length* field is zero.

Message data extension pointer

A pointer to the area to receive the message data extension. This field is ignored if the *max extension data length* field is zero.

Message format information

A 32 byte field that contains either compare data or two 16 byte fields which contain information related to a return type message.

•

- If *message type* is not a *return* or *return/transfer control* message, this field is defined as compare data used to determine which exception handler is given control. Up to 32 bytes may be specified.
- If *message type* is a *return* or *return/transfer control* message, then the first 16 bytes of this field are defined as a *system pointer* to an *program* object, or else a *procedure pointer* to a bound program procedure. The last 16 bytes of the field are reserved for future use.

This value is ignored if the message does not represent an *exception, return* or *return/transfer control* message.

The template identified by *operand 3* specifies the source invocation of the message. This operand can be null (which indicates the requesting invocation is to be used for the Source Invocation) or specify either an Invocation pointer to an invocation, a null pointer value (which indicates the current invocation), or a pointer to a process queue space. It must be 16-byte aligned with the following format.

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Source invocation offset	Bin(4)
4	4	Originating invocation offset	Bin(4)
8	8	Invocation range	Bin(4)
12	C	Reserved (binary 0)	Char(4)
16	10	Source invocation/process queue space pointer	System pointer or Invocation pointer
32	20	Reserved (binary 0)	Char(16)
48	30	—- End —-	

Source invocation offset

A signed numerical value indicating an invocation relative to the invocation located by the *source invocation pointer*. A value of zero denotes the invocation addressed by the *source invocation pointer*, with increasingly positive numbers denoting increasingly later invocations in the stack, and increasingly negative numbers denoting increasingly earlier invocations in the stack. If a process queue space is specified as the message source, then the only valid values for this field are 0, -1 and -2. A value of -1 indicates to materialize from the external queue of the process queue space. A zero value indicates to materialize from the message log of the process queue space. A value of -2 indicates to attempt to locate the message using the *message reference index* supplied in operand 4 without regard to the queue space queue that the message resides on. Only unanswered *inquiry* messages, *return* messages, and *return/transfer control* messages can be materialized in this fashion. Other values result in a *scalar value invalid* (hex 3203) exception being signaled.

If the invocation identified by this offset does not exist in the stack, a *scalar value invalid* (hex 3203) exception will be signaled.

Originating invocation offset

Specifies a displacement from the invocation executing this instruction and must be zero (which indicates the current invocation) or negative (which indicates an older invocation). The invocation identified is used as the source for all authorization checks (environment authority to an invocation or authority to a process queue space). If the *originating invocation offset* is non-zero, then the invocation executing this instruction must be authorized to the originating invocation identified.

If the invocation identified by this offset does not exist in the stack or the value is greater than zero, a *scalar value invalid* (hex 3203) exception will be signaled.

Invocation range

A signed numerical value indicating the number of invocations in the range in addition to the invocation identified by the *source invocation pointer*. If a *process queue space pointer* is provided, this value must be zero.

The sign of the *invocation range* determines the direction of the additional invocations. A positive number specifies a range encompassing newer invocations, while a negative number specifies a range encompassing older invocations.

It is not an error if this value specifies a range greater than the number of existing invocations in the specified direction. The materialization will stop after the last invocation is encountered.

Source invocation pointer

An *invocation pointer* to an invocation. If a null pointer value, then the current invocation is indicated.

If the invocation identified does not exist in the stack or is invalid for this operation, an *invalid invocation address* (hex 1603) exception will be signaled.

Process queue space pointer

A *system pointer* to a *process queue space* object.

The template identified by operand 4 must be 16-byte aligned. Following is the format of the message selection template:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Starting message reference index	Char(4)
4	4	Ending message reference index	Char(4)
8	8	Number of selection criteria	Bin(2)
10	A	Reserved (binary 0)	Char(6)
16	10	Selected message reference index	Char(4)
		Output by the instruction	
20	14	Selected message count	Bin(4)
		Output by the instruction	
24	18	Status change count	Bin(4)
		Ignored by the instruction	
28	1C	Moved message count	Bin(4)
		Ignored by the instruction	
32	20	Selection criterion	[*] Char(32)
32	20	Selection type	Char(1)

Offset Dec	Hex	Field Name	Data Type and Length
			Hex 00 = Select based on message status
			Hex 01 = Select based on message ID
			Hex 02 = Select based on interrupt class
			Hex 03 = Select based on invocation mark
			Hex 04 = Select based on activation group mark
			Hex 07 = Select based on thread ID
			Hex 08 = Select based on 8-byte invocation mark
			Hex 09 = Select based on 8-byte activation group mark
33	21	Reserved (binary 0)	Char(1)
34	22	Selection action	Char(2)
34	22	Reject criterion	Bit 0
		0 = Select message if criterion is satisfied	
		1 = Reject message if criterion is satisfied	
34	22	Reject message	Bit 1
		0 = Do not reject message if criterion is not satisfied	
		1 = Reject message if criterion is not satisfied	
34	22	Reject satisfaction	Bit 2
		0 = Accept satisfaction of criterion	
		1 = Reject satisfaction of criterion	
34	22	Reserved (binary 0)	Bits 3-15
36	24	Message type mask	Char(4)
40	28	Selection criterion information	Char(24)
		If the *selection type* is *message status*	
40	28	Message status mask	Char(8)
48	30	Message status complement	Char(8)
56	38	Reserved (binary 0)	Char(8)
64	40	—- End of message status —-	
		If the *selection type* is *message ID*	
40	28	Message ID	Char(7)
47	2F	Reserved (binary 0)	Char(17)
64	40	—- End of message ID —-	
		If the *selection type* is *interrupt class*	
40	28	Interrupt class mask	Char(8)

Offset Dec	Offset Hex	Field Name	Data Type and Length		
48	30		Interrupt class complement		Char(8)
56	38		Reserved (binary 0)		Char(8)
64	40		—- End of interrupt class —-		
			If the *selection type* is *invocation mark*		
40	28		Invocation mark		UBin(4)
44	2C		Reserved (binary 0)		Char(20
64	40		—- End of invocation mark —-		
			If the *selection type* is *activation group mark*		
40	28		Activation group mark		UBin(4)
44	2C		Reserved (binary 0)		Char(20
64	40		—- End of activation group mark —-		
			If the *selection type* is *thread ID*		
40	28		Thread ID		Char(8)
48	30		Reserved (binary 0)		Char(16
64	40		—- End of thread ID —-		
			If the *selection type* is *8-byte invocation mark*		
40	28		Invocation mark		UBin(8)
			For Non-Bound programs, the following datatype should be used:		
40	28			Invocation mark (Non-Bound program)	Char(8)
48	30		Reserved (binary 0)		Char(16
64	40		—- End of 8-byte invocation mark —-		
			If the *selection type* is *8-byte activation group mark*		
40	28		Activation group mark		UBin(8)
			For Non-Bound programs, the following datatype should be used:		
40	28			Activation group mark (Non-Bound program)	Char(8)
48	30		Reserved (binary 0)		Char(16
64	40		—- End of 8-byte activation group mark —-		
*	*	—- End —-			

Starting and ending message reference index

Messages in the specified range of index values are examined either until one of the selection criteria has been satisfied or all queues specified have been searched. The direction of search is determined by the relative values of *starting message reference index* and *ending message reference index*. If the former value is smaller, then the search direction is in numerically (and chronologically) increasing order, while if the latter value is smaller the search direction is in the opposite direction.

Messages are examined starting with the message identified by *starting message reference index*, or if no such message exists in the queue, starting with the closest existing message in the direction of the search.

For the current process, queue space messages will only be selected for the current thread. To select messages for another thread, a thread selection criteria must be used.

For a different process queue space, all messages for all threads will be selected.

If *operand 3* specifies a system pointer to a queue space and a *source invocation offset* of -2, then the *starting and ending message reference indices* must be equal, otherwise a *scalar value invalid* (hex 3203) exception will be issued.

Number of selection criteria

A numerical value that specifies how many selection criteria fields are supplied. If *number of selection criteria* has a value of zero, then the first message in the index range will be materialized.

Selected message reference index

This value returns the message reference index of the message materialized. Zero is returned if no message satisfies the criteria.

Selected message count

This value indicates the number of messages selected: zero for no messages found, one if a message was found.

Selection criterion

This field contains the data used to select messages from a queue space. There is a variable number of criteria present in the template (the number present is in the *number of selection criteria* field). Each selection criterion may select a message, reject it, or take no action. Successive selection criteria are applied to each message until it is selected or rejected, or until selection criteria have been exhausted (in which case selection is the default).

Selection type

This field indicates the format of the selection criterion and what field in the message is compared.

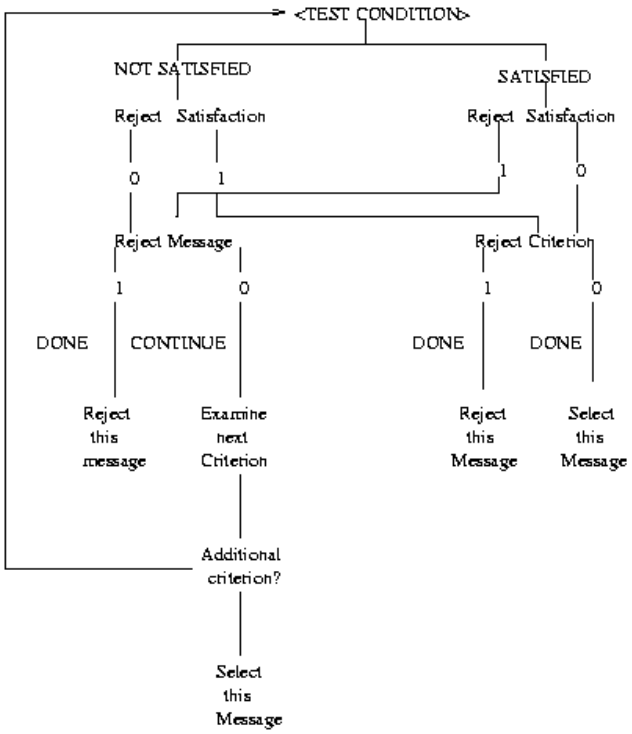
Selection action

A bit-significant value indicating what actions to perform during the selection criteria processing.

•

- **Reject criterion** - If this bit is 1, a message is rejected if the selection criterion is satisfied. If this bit is 0, a message is selected when the selection criterion is satisfied.
- **Reject message** - If this bit is 1, a message is rejected if the selection criteria is not satisfied. If this bit is 0, no action is taken when the selection criteria is not satisfied. Processing continues with the next selection criterion.
- **Reject satisfaction** - If this bit is 0, the satisfaction of the criterion is accepted. If this bit is 1, a satisfied criterion will become not satisfied and a not satisfied criterion will become satisfied.

The following figure illustrates how messages are selected based on the *selection action* criterion.



Message type mask

A bit-significant value indicating types of messages that should be examined during selection criteria processing. The first 31 bits correspond to message types hex 00 through hex 1E respectively. The 32nd bit (bit 31) corresponds to all message types greater than hex 1E.

Message status mask and message status complement

These are bit-significant values indicating the message status attributes that will allow a message to be selected. These values are used to test a message as follows:

The *message status complement* is bit-wise exclusive-ORed with the message status value of a message. The result of this is then bit-wise ANDed with the *message status mask*. If the result is all 0 bits, the message does not satisfy this selection criterion. If the result is not all 0 bits, the message satisfies the selection criterion.

Message ID

A character value that is compared to the message ID of a message. If the values are equal the selection is satisfied.

Interrupt class mask and interrupt class complement

These are bit-significant values indicating the interrupt class attributes that will allow a message to be selected. These values are used to test a message as follows:

The *interrupt class complement* is bit-wise exclusive-ORed with the interrupt class value of a message. The result of this is then bit-wise ANDed with the *interrupt class mask*. If the result is all 0 bits, the message does not satisfy this selection criterion. If the result is not all 0 bits, the message satisfies the selection criterion.

Invocation mark

A binary number that is compared to the invocation mark of a message. If the values are equal the selection is satisfied. The value returned in the 4-byte invocation mark may have wrapped.

Activation group mark

A binary number that is compared to the activation group mark of a message. If the values are equal the selection is satisfied. The value returned in the 4-byte activation group mark may have wrapped.

Thread ID

A binary number that is compared to the thread ID of a message. If the values are equal, the selection is satisfied.

Note: 0 is not a valid thread ID and can be used to select all messages.

Warning: Temporary Level 3 Header

Authorization Required

The following algorithm is used to determine authorization.

1. The invocation which invoked the MATPRMSG instruction must have authority to the invocation identified as the Source Invocation.
2. The Originating Invocation must have authority to the invocation identified as the Source Invocation or to the invocation directly called by the Source invocation.

If any of the authority checks fail then an *activation group access violation* (hex 2C12) exception will be signaled.

•

- Operational
  –
  – Operand 3
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Operand 3
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

16 Exception Management

1603 Invalid Invocation Address

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2C Program Execution

2C12 Activation Group Access Violation

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

46 Queue Space

4601 Queue Space Not Associated with the Process

Materialize Process Mutex (MATPRMTX)

Bound program access
Built-in number for MATPRMTX is 164. MATPRMTX (operand1 : address operand2 : address of system pointer OR address of space pointer(16) OR null pointer value operand3 : address of unsigned binary(4) value OR null pointer value)

Note: The term ″mutex″ in this instruction refers to a ″pointer-based mutex″.

Description: The mutex lock and/or wait status of the thread or threads specified by operand 2 is materialized into the receiver space specified by operand 1. Operand 2 is the address of a process control space system pointer, the address of a space pointer or a null pointer value. If operand 2 is the address of a process control space system pointer, mutex status is returned for the initial thread of the process, or optionally, mutex status is returned for multiple threads in the process. If operand 2 is the address of a

space pointer to a thread handle, mutex status is returned for the thread specified by the thread handle, or optionally, mutex status is returned for multiple threads in the same process as the specified thread. If operand 2 is a null pointer value, mutex status is returned for the issuing thread, or optionally, mutex status is returned for multiple threads in the same process as the issuing thread. If operand 2 references a process that is not a valid process, a *process control space not associated with a process* (hex 2802) exception is signaled. If operand 2 references a thread handle that is not valid, a *thread handle not associated with an active thread* (hex 2804) exception is signaled. For the specified thread or threads, the materialization identifies each mutex for which the thread holds a mutex lock and/or for which the thread is waiting, if any. The use of the MATPRMTX instruction to obtain information for threads in another process requires that the issuing thread be contained in the process that initiated the specified thread's process, or requires that the issuing thread have process control special authorization defined in its user profile or in a currently adopted user profile. If the issuing thread does not have the necessary authorization, a *special authorization required* (hex 0A04) exception is signaled. The space pointed to by operand 3 is a 4-byte unsigned binary field used to indicate the type of information that should be returned by this instruction.

The **materialization options** value referenced by operand 3 has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Mutex reference option	Bit 0
		0 = Return addresses of mutexes	
		1 = Return replicas of mutexes	
0	0	Materialize threads option	Bit 1
		0 = Materialize only specified thread	
		1 = Materialize multiple threads in specified thread's process	
0	0	Mutex attributes option	Bit 2
		0 = Do not return additional mutex attributes	
		1 = Return additional mutex attributes	
0	0	Reserved (binary 0)	Bit 3
0	0	Thread status option	Bit 4
		0 = Materialize holding and waiting threads	
		1 = Materialize only waiting threads	
0	0	Reserved (binary 0)	Bits 5-31
4	4	—- End —-	

If **mutex reference option** is set to *return addresses of mutexes*, then the materialization template will contain the addresses of the mutexes which are being described. If *mutex reference option* is set to *return replicas of mutexes*, then the materialization template will contain replicas of the mutexes which are being described. A replica is a mechanism used to obtain mutex information for mutexes that reside outside the addressability of the issuing thread. The address of a mutex or the address of a replica can be passed to the MATMTX instruction for further materialization. However, a replica can only be used to materialize a mutex, it cannot be used to perform any operation on the mutex that could change its state.

If the *mutex reference option* is set to *return addresses of mutexes*, only those mutexes that the issuing thread has addressability to can be materialized. If the *mutex reference option* is set to *return replicas of mutexes*, all mutexes for the specified thread(s) can be materialized. Addressable mutexes are mutexes that exist outside of a teraspace, or exist in the teraspace of the issuing thread. Mutexes that exist in teraspace other than the teraspace of the issuing thread are not eligible to be materialized if the *mutex reference option* is set to *return addresses of mutexes*, but are eligible to be materialized if the *mutex reference option* is set to *return replicas of mutexes*. Copies of mutexes are not materialized, regardless of addressability. See the CRTMTX instruction for additional information regarding mutex copies.

The **materialize threads option** field is used to select if mutex status is to be returned for only the specified thread, or for each thread in the same process as the specified process or thread. If *materialize threads option* is set to *materialize only specified thread*, then mutex status can be returned for only the specified thread. If *materialize threads option* is set to *materialize multiple threads in specified thread's process*, then mutex status can be returned for each thread in the same process as the specified process or thread. For a specified thread, or each of multiple threads, mutex status can only be returned if the thread meets the criteria specified by the *thread status option*.

The **mutex attributes option** field is used to select whether or not additional mutex attributes are to be returned. If *mutex attributes option* is set to *do not return additional mutex attributes*, then a standard materialization template is used. If *mutex attributes option* is set to *return additional mutex attributes*, then a materialization template with extended *mutex descriptors* is used.

The **thread status option** field is used to selectively determine if a thread can be materialized based on its mutex status. If *thread status option* is set to *materialize holding and waiting threads*, then status can be returned for the specified thread or threads if the thread is holding a mutex lock and/or is waiting on a mutex. If *thread status option* is set to *materialize only waiting threads*, then status can be returned for the specified thread or threads only if the thread is waiting on a mutex.

If operand 3 is a null pointer value, the default *materialization options* (binary 0) are used. All values other than those specifically defined for *materialization options* are reserved and will cause a *scalar value invalid* (hex 3203) exception to be generated.

The materialization template identified by operand 1 must be 16-byte aligned. If the materialization template is not properly aligned, a *boundary alignment* (hex 0602) exception is signaled. The format of the information returned in the materialization template is different, depending on the *materialization options* selected.

The materialization template has the following standard format when *materialize threads option* is set to *materialize only specified thread* and *mutex attributes option* is set to *do not return additional mutex attributes*:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Number of mutex lock entries	Bin(4)	
12	C	Reserved (binary 0)	Char(4)	
16	10	Mutex descriptors (repeated for each mutex currently held locked or waited for by the thread)	[*] Char(32)	
16	10		Mutex	Space pointer or o
32	20		Mutex state being described	Char(1)
			Hex 00 = The mutex is held by the thread	
			Hex 01 = The thread is waiting to acquire the mutex	
33	21		Reserved (binary 0)	Char(15)
*	*	—- End —-		

The first 4 bytes of the materialization template identifies the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization template identifies the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver that can be used to completely fill *mutex descriptors*. Partial descriptors are not returned. If the *number of bytes provided* would cause the storage boundary of the space provided for the receiver to be exceeded, and if the *number of bytes available* would actually exceed this boundary, then a *space addressing violation* (hex 0601) exception is signaled. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception described previously.

The **number of mutex lock entries** is the number of mutexes the materialized thread currently holds locked and/or is waiting for.

The **mutex descriptors** identify the mutexes the materialized thread either holds (acquired the lock on) or is waiting for. Depending on the setting of *mutex reference option*, the **mutex** field contains either the address of the mutex being described or a replica of the mutex being described. The **mutex state being described** field identifies how this mutex relates to the thread being materialized.

The materialization template has the following extended format when *materialize threads option* is set to *materialize only specified thread* and *mutex attributes option* is set to *return additional mutex attributes*:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Number of mutex lock entries	Bin(4)	
12	C	Reserved (binary 0)	Char(4)	
16	10	Mutex descriptors (repeated for each mutex currently held locked or waited for by the thread)	[*] Char(112)	
16	10		Mutex	Space pointer o
32	20		Mutex state being described	Char(1)
			Hex 00 = The mutex is held by the thread	
			Hex 01 = The thread is waiting to acquire the mutex	
33	21		Reserved (binary 0)	Char(15)
48	30		Mutex name	Char(16)
64	40		Mutex holder process ID	Char(30)
94	5E		Reserved (binary 0)	Char(2)
96	60		Mutex holder thread ID	Char(8)
104	68		Mutex holder unique thread value	Char(8)
112	70		Number of waiters	Bin(4)
116	74		Reserved (binary 0)	Char(12)
*	*	—- End —-		

The contents of the template fields are as defined for the previous template(s), unless specifically defined or redefined as follows:

The **mutex name** field contains the name of the mutex. The name is left-justified and padded to the right with blanks. If the mutex was created using a null-terminated name string, the name materialized with this instruction is null-terminated instead of padded with blanks. If the mutex was created without a name, this field will contain the character string ″UNNAMED_″ followed by the first 8 characters of the

program which created the mutex. The **mutex holder process ID** is the name of the process containing the thread that holds the mutex lock. The name returned here is the 30-character Process Control Space (PCS) name. The **mutex holder thread ID** contains a process specific value that identifies the thread within the process that holds the mutex lock. The **mutex holder unique thread value** contains a system-wide unique value that identifies the specific thread that holds the mutex lock. This field cannot be used as input on any other MI instruction, but may be useful for debug purposes. The **number of waiters** is the number of threads that are currently waiting for the mutex to be unlocked.

The materialization template has the following standard format when *materialize threads option* is set to *materialize multiple threads in specified thread's process* and *mutex attributes option* is set to *do not return additional mutex attributes*:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Number of threads in process	UBin(4)	
12	C	Number of thread mutex descriptors	UBin(4)	
16	10	Thread mutex descriptors (repeated for each mutex currently held locked or waited for by each identified thread)	[*] Char(48)	
16	10		Identified thread ID	Char(8
24	18		Number of descriptors for identified thread	UBin(4
28	1C		Descriptor entry number for identified thread	UBin(4
32	20		Mutex descriptor	Char(3
32	20		Mutex	
48	30		Mutex state being described **Hex 00 =** The mutex is held by the thread **Hex 01 =** The thread is waiting to acquire the mutex	
49	31		Reserved (binary 0)	
*	*	—- End —-		

The contents of the template fields are as defined for the previous template(s), unless specifically defined or redefined as follows:

The **number of threads in process** is the total number of active threads in the process found at the time of materialization. The actual number of threads materialized may be less than this number since only threads that are associated with at least one mutex will have their mutex information described. A thread has a mutex association if it has at least one mutex locked and/or is waiting on a mutex. The actual number of threads materialized may also be less than this number if the *number of bytes provided* is insufficient to contain all of the mutex information for each thread in the process having a mutex association.

The **number of thread mutex descriptors** is the combined total number of *thread mutex descriptors* that were actually returned for all the threads in the process. The number of descriptors returned for each thread varies depending on the number of mutexes an individual thread is associated with, that is, the number of mutexes the thread has locked and/or is waiting for. If the *number of bytes provided* is insufficient to contain all of the mutex information for all of the threads in the process, then as many mutex descriptors are returned for as many threads as possible. Partial descriptors are not returned.

The **thread mutex descriptors** identify the threads that are associated with one or more mutexes, and describe each mutex associated with each thread. One *thread mutex descriptor* is returned for each mutex associated with a given thread, so the same thread can have multiple descriptors. The selection of descriptors to be returned for a specified thread or threads is determined by the mutex association criteria specified by the *thread status option*. The **identified thread ID** field contains a process specific value that identifies the thread that is associated with a particular descriptor. The **number of descriptors for identified thread** field contains the number of possible descriptors that can be returned for the *identified thread*. The **descriptor entry number for identified thread** field uniquely identifies each descriptor returned for the *identified thread*. For each descriptor associated with the same thread, the *identified thread ID* and *number of descriptors for identified thread* fields will contain the same values, but the *descriptor entry number* will be sequentially different. The combination of these three fields effectively organizes the mutex information returned for each thread into a ″descriptor M of N″ format where ″M″ is the *descriptor entry number for identified thread* and ″N″ is the *number of descriptors for identified thread*.

The materialization template has the following extended format when *materialize threads option* is set to *materialize multiple threads in specified thread's process* and *mutex attributes option* is set to *return additional mutex attributes*:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin
4	4		Number of bytes available for materialization	Bin
8	8	Number of threads in process	UBin(4)	
12	C	Number of thread mutex descriptors	UBin(4)	
16	10	Thread mutex descriptors (repeated for each mutex currently held locked or waited for by each identified thread)	[*] Char(128)	
16	10		Identified thread ID	Cha
24	18		Number of descriptors for identified thread	UBi
28	1C		Descriptor entry number for identified thread	UBi
32	20		Mutex descriptor	Cha
32	20		Mutex	
48	30		Mutex state being described **Hex 00 =** The mutex is held by the thread **Hex 01 =** The thread is waiting to acquire the mutex	
49	31		Reserved (binary 0)	
64	40		Mutex name	
80	50		Mutex holder process ID	
110	6E		Reserved (binary 0)	
112	70		Mutex holder thread ID	
120	78		Mutex holder unique thread value	
128	80		Number of waiters	
132	84		Reserved (binary 0)	
*	*	— End —		

The contents of the template fields are as defined for the previous template(s).

Warning: Temporary Level 3 Header

Authorization Required

•

• Process control special authorization

  –

  – For materializing a thread in a process other than the process containing the thread issuing this instruction.

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A04 Special Authorization Required

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2404 Pointer Not Resolved

28 Process/Thread State

2802 Process Control Space Not Associated with a Process

2804 Thread Handle Not Associated with an Active Thread

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Process Record Locks (MATPRECL)

Op Code (Hex)	Operand 1	Operand 2
031E	Receiver	Process selection template

Operand 1: Space pointer.

Operand 2: Space pointer.

Bound program access
Built-in number for MATPRECL is 52. MATPRECL (receiver : address process_selection_template : address)

Description: Data space record locks for a process identified in the *process selection template* specified by operand 2 are materialized into the *receiver* identified by operand 1. The materialization identifies each data space record lock which is either held by the process or is waited for by a thread within the process.

If the process control space (PCS) pointer is null or all zeros, the lock activity for the process containing the current thread is materialized.

The *process selection template* identified by operand 2 must be 16-byte aligned. The format of the *process selection template* is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Process selection	Char(16)	
0	0		Process descriptions	System pointer
16	10	Lock selection	Char(1)	
16	10		Materialize held locks	Bit 0
			0 = Do not materialize	
			1 = Materialize	
16	10		Materialize locks waited for	Bit 1
			0 = Do not materialize	
			1 = Materialize	
16	10		Reserved	Bits 2-7
17	11	Template options	Char(1)	
17	11		Format for number of locks	Bit 0
			1 = Use Bin(4) for number of locks	
			0 = Use Bin(2) for number of locks	
17	11		Reserved	Bits 1-7
18	12	Reserved	Char(6)	
24	18	—- End —-		

The **process descriptions** must be a system pointer to a process control space (PCS) or a null pointer value.

Both of the fields specified under **lock selection** are bits which determine the locks to be materialized. If the **materialize held locks** is *materialize*, any data base record lock held by the process is materialized. If the **materialize locks waited for** is *materialize*, any data base record lock a thread of the process is waiting for is materialized.

The **format for number of locks** bit determines the format of the *number of lock held descriptions* and *number of locks waited for descriptions* fields in the materialization template. If the bit is set on then Bin(4) counts are used, else Bin(2) counts are used.

The materialization template identified by operand 1 must be 16-byte aligned. The format of the materialization is as follows:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4
4	4		Number of bytes available for materialization	Bin(4
8	8	Materialization data (2 possible formats)	Char(8)	
			If format for number of locks bit=1	
8	8		Number of lock held descriptions	Bin(4
12	C		Number of lock waited for desscriptions	Bin(4
16	10		—- End of bit=1 —-	
			If format for number of locks bit=0	
8	8		Number of lock held descriptions	Bin(2
10	A		Number of lock waited for descriptions	Bin(2
12	C		Reserved	Char
16	10		—- End of bit=0 —-	
16	10	Locks held descriptions (repeated *number of lock held descriptions* times)	[*] Char(32)	
16	10		Data space	Syste
32	20		Relative record number	UBin
36	24		Lock state	Char
			Hex 30 = DLWK (Database lock weak) lock state	
			Hex C0 = DLRD (Database lock read) lock state	
			Hex F8 = DLUP (Database lock update) lock state	
			All other values are reserved.	
37	25		Lock holder information	Char
37	25		Lock scope object type	
			0 = Process control space	
			1 = Transaction control structure	
37	25		Lock scope	
			0 = Lock is scoped to the *lock scope object type*	
			1 = Lock is scoped to the thread	
37	25		Reserved (binary 0)	
38	26		Reserved	Char
40	28		Thread ID	Char
*	*	Lock waited for descriptions (repeated *number of lock waited for descriptions* times)	[*] Char(32)	
*	*		Data space	Syste
*	*		Relative record number	UBin
*	*		Lock state requested	Char

Offset Dec	Offset Hex	Field Name	Data Type and Length
			Hex 30 = DLWK (Database lock weak) lock state
			Hex C0 = DLRD (Database lock read) lock state
			Hex F8 = DLUP (Database lock update) lock state
			All other values are reserved.
*	*		Lock waiter information Char(1)
*	*		Lock scope object type
			0 = Process control space
			1 = Transaction control structure
*	*		Lock scope
			0 = Lock is scoped to the *lock scope object type*
			1 = Lock is scoped to the thread
*	*		Reserved (binary 0)
*	*		Reserved Char(2)
*	*		Thread ID Char(8)
*	*	—- End —-	

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, the excess bytes are unchanged. No exceptions are signaled in the event that the *receiver* contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception described previously.

The **number of lock held descriptions** contains the number of locks held by the process. One system pointer to the **data space**, **relative record number** in the data space, and **lock state** is materialized in the area identified as **lock held descriptions** for each lock. When **lock scope** has a value of *lock is scoped to the thread*, the **thread ID** field identifies the thread that holds the lock. Otherwise it is set to binary 0. These fields contain data only if *materialize held locks* is *materialize*.

A database weak record lock is only acquired thread-scoped and it only conflicts with update record locks which are thread-scoped to a different thread. The weak record lock does not conflict in any other situation.

The **number of lock waited for descriptions** contains the number of locks that the process is waiting for. One system pointer to the **data space**, **relative record number** in the data space, and **lock state requested** is materialized in the area identified as **lock waited for descriptions** for each lock waited for. The **thread ID** field identifies the thread that is waiting for the lock, regardless of the **lock scope** value. These fields contain data only if *materialize locks waited for* is *materialize*.

If Bin(2) fields are requested for the *number of lock held descriptions* and *number of lock waited for descriptions*, then the maximum number that can be returned in each count is 32,767. If the actual number is greater than 32,767 for a count then that count will be set to 32,767, only the first 32,767 locks will be materialized, and no exception will be signaled.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Program (MATPG)

Op Code (Hex)	Operand 1	Operand 2
0232	Attribute receiver	Program

Operand 1: Space pointer.

Operand 2: System pointer.

Bound program access
Built-in number for MATPG is 31. MATPG (attribute_receiver : address program : address of system pointer)

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The non-bound *program* identified by operand 2 is materialized into the template identified by operand 1.

Operand 2 is a system pointer that identifies the *program* to be materialized. The program identified by operand 2 must be a non-bound program. Otherwise, a *program not eligible for operation* (hex 220A) exception will be signalled. The values in the materialization relate to the current attributes of the materialized program.

This instruction does not process teraspace addresses used for its operands, nor used in any space pointer contained in a template. Any teraspace address use will cause an *unsupported space use* (hex 0607) exception to be signaled, whether the issuing program is teraspace capable or not.

The template identified by operand 1 must be 16-byte aligned.

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Template size specification	Char(8)	
0	0		Number of bytes provided	Bin(4)
4	4		Number of bytes available for materialization (used only when the program is materialized)	Bin(4) +
8	8	Program identification	Char(32)	
8	8		Type	Char(1) +
9	9		Subtype	Char(1)
10	A		Name	Char(30)
40	28	Program creation options	Char(4)	
40	28		Existence attributes	Bit 0
			0 = Temporary	
			1 = Permanent	
40	28		Space attribute	Bit 1
			0 = Fixed-length	
			1 = Variable-length	
40	28		Initial context	Bit 2
			0 = Do not insert addressability into context	
			1 = Insert addressability into context	
40	28		Access group creation	Bit 3

Offset Dec	Offset Hex	Field Name	Data Type and Length		
			0 =	Do not create as a member of an access group	
			1 =	Create as a member of an access group	
40	28		Reserved (binary 0)		Bits 4-12
40	28		Initialize space		Bit 13
			0 =	Initialize	
			1 =	Do not initialize	
40	28		Automatically extend space		Bit 14
			0 =	No	
			1 =	Yes	
40	28		Associated space hardware storage protection level		Bits 15-16
			00 =	Reference and modify allowed for user state programs	
			01 =	Only reference allowed for user state programs	
			11 =	No reference or modify allowed for user state programs	
40	28		Reserved (binary 0)		Bits 17-31
44	2C	Reserved (binary 0)	Char(4)		
48	30	Size of space	Bin(4)		
52	34	Initial value of space	Char(1)		
53	35	Performance class	Char(4)		
53	35		Obsolete This field is no longer used and will be ignored.		Bit 0 +
53	35		Reserved (binary 0)		Bits 1-4
53	35		Main storage pool selection		Bit 5
			0 =	Process default main storage pool is used for object.	
			1 =	Machine default main storage pool is used for object.	
53	35		Transient storage pool selection		Bit 6
			0 =	Default main storage pool (process default or machine default as specified for main storage pool selection) is used for object.	
			1 =	Transient storage pool is used for object.	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
53	35		Block transfer on implicit access state modification	Bit 7
			0 = Transfer the minimum storage transfer size for this object.	
			1 = Transfer the machine default storage transfer size for this object.	
53	35		Reserved (binary 0)	Bits 8-31
57	39	Reserved (binary 0)	Char(7)	
64	40	Context	System pointer	
80	50	Access group	System pointer	
96	60	Program attributes	Char(2)	
96	60		Adopted user profile	Bit 0
			0 = No adoption of user profile.	
			1 = Adopt program owner's user profile on invocation.	
96	60		Array constraint	Bit 1
			0 = Arrays are constrained.	
			1 = Arrays are unconstrained. The predictability of the results of references outside the bounds of arrays are determined by the *type of unconstrained arrays* field.	
96	60		String constraint	Bit 2
			0 = Strings are constrained.	
			1 = Strings are not constrained.	
96	60		Obsolete	Bit 3 +
96	60		Adopted user profile propagation	Bit 4
			0 = Adopted user profile authorities are not propagated to external invocations.	
			1 = Adopted user profile authorities are propagated to all subinvocations.	
96	60		Static storage	Bit 5
			0 = Initialize storage to binary 0.	
			1 = Do not initialize storage to binary 0.	
96	60		Automatic storage	Bit 6
			0 = Initialize storage to binary 0.	
			1 = Do not initialize storage to binary 0.	
96	60		Associated journal entry	Bit 7

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = Program name is recorded in journal entries	
			1 = Program name is not recorded in journal entries	
96	60		Update PASA stack This field is now obsolete. It will be ignored.	Bit 8 +
96	60		Suppress decimal data exception	Bit 9
			0 = Exception is not to be suppressed	
			1 = Exception is to be suppressed	
96	60		Template extension existence	Bit 10
			0 = Template extension does not exist	
			1 = Template extension exists	
96	60		Suppress previously adopted user profiles	Bit 11
			0 = Do not suppress previously adopted user profiles	
			1 = Suppress previously adopted user profiles	
96	60		Template version	Bits 12-15
			0000 = Version 0	
			0001 = Version 1	
			0010 through 1111 reserved	
98	62	Code generation options	Char(1)	
98	62		Performance optimization	Bit 0
			0 = No optimization	
			1 = Perform optimization	
98	62		Space pointer machine objects	Bit 1
			0 = Disallow space pointer machine objects in ODV component	
			1 = Allow space pointer machine objects in ODV component	
98	62		Coincident operand overlap	Bit 2
			0 = Do not assume coincident operand overlap	
			1 = Assume coincident operand overlap	
98	62		Reserved (binary 0)	Bits 3-4
98	62		Teraspace capable	Bit 5
			0 = Do not generate teraspace capable program	
			1 = Generate teraspace capable program	
98	62		Executable part compression	Bit 6

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = Do not compress executable part	
			1 = Compress executable part	
98	62		Observation part compression	Bit 7
			0 = Do not compress observation part	
			1 = Compress observation part	
99	63	Observation attributes	Char(1)	
		For bits 0 through 5:		
		1 = The corresponding template component is materializable		
		0 = The corresponding template component is not materializable		
99	63		Instruction stream	Bit 0
99	63		ODT Directory Vector (ODV)	Bit 1
99	63		ODT Entry String (OES)	Bit 2
99	63		Breakpoint Offset Mapping (BOM) table	Bit 3
99	63		Symbol table	Bit 4
99	63		Object Mapping Table (OMT)	Bit 5
		For bits 6 and 7:		
		1 = The corresponding performance measurement is prevented		
		0 = The corresponding performance measurement is allowed		
99	63		Prevent performance measurements on entry/exit	Bit 6
99	63		Prevent performance measurements on CALLX	Bit 7
100	64	Size of static storage	UBin(4)	
104	68	Size of automatic storage	UBin(4)	
108	6C	Number of instructions (1)	UBin(2)	
		For *version number* = hex 0000, this field indicates the number of instructions. For *version number* = hex 0001, this field is reserved (binary 0).		
110	6E	Number of ODV entries (1)	Bin(2)	
		For *version number* = hex 0000, this field indicates the number of ODV entries. For *version number* = hex 0001, this field is reserved (binary 0).		
112	70	Offset (in bytes) from beginning of template to the instruction stream component	Bin(4)	
116	74	Offset (in bytes) from beginning of template to the ODV component	Bin(4)	
120	78	Offset (in bytes) from beginning of template to the OES component	Bin(4)	
124	7C	Length of breakpoint offset mapping table entry	Bin(4)	
128	80	Length of breakpoint offset mapping table component	Bin(4)	
132	84	Offset (in bytes) from beginning of template to the BOM table	Bin(4)	
136	88	Length of symbol table entry	Bin(4)	
140	8C	Length of symbol table component	Bin(4)	

Offset Dec	Offset Hex	Field Name	Data Type and Length
144	90	Offset (in bytes) from beginning of template to the Symbol table	Bin(4)
148	94	Offset (in bytes) from beginning of template to the object mapping table (OMT) component	Bin(4) +
152	98	Number of instructions (2) For *version number* = hex 0001, this field indicates the number of instructions. For *version number* = hex 0000, this field is reserved (binary 0).	Bin(4)
156	9C	Number of ODV entries (2) For *version number* = hex 0001, this field indicates the number of ODV entries. For *version number* = hex 0000, this field is reserved (binary 0).	Bin(4)
160	A0	Template extension This extension exists only when the *template extension existence* bit is 1.	Char(64)
160	A0		Extended program attributes — Char(4)
160	A0		Type of unconstrained arrays — Bit 0 **0 =** Not fully unconstrained. If arrays are unconstrained, unpredictable results may occur when accessing array elements outside the declared bounds of the array. **1 =** Fully unconstrained. Predictable results will occur when accessing array elements outside the declared bounds of the array. See the paragraph below describing array constrainment for details.
160	A0		Suppress binary size exception — Bit 1 **0 =** Exception is not to be suppressed **1 =** Exception is to be suppressed
160	A0		Create program for previous mandatory release — Bit 2 **0 =** Create the program to run on the current release **1 =** Create the program to run on the previous mandatory release
160	A0		Collect object usage data for program — Bit 3 **0 =** Collect the object usage data **1 =** Do not collect the object usage data
160	A0		Scope of resources — Bit 4

Offset Dec	Offset Hex	Field Name	Data Type and Length		
			0 =	Resources are scoped to an invocation of this program.	
			1 =	Resources are scoped to a program previous to this one in the invocation stack.	
160	A0		Reserved (binary 0)		Bits 5-
164	A4		Language version, release, and modification level	Char(2)	
164	A4		Reserved		Bits 0-
164	A4		Version		Bits 4-
164	A4		Release		Bits 8-
164	A4		Mod level		Bits 1
166	A6		Breakpoint offset mapping table data	Char(1)	
166	A6		BOM table flags		Char(
166	A6			Use new BOM table format	Bit 0
166	A6			User data5A	Bits 1-
167	A7		User data5B	Char(7)	
174	AE		Version, release, and modification level this program is being created for	Char(2)	
174	AE		Reserved		Bits 0-
174	AE		Version		Bits 4-
174	AE		Release		Bits 8-
174	AE		Mod level		Bits 1
176	B0		Data required for machine retranslation	Char(1)	
176	B0		All data required for machine retranslation is present		Bit 0 -
			0 =	No	
			1 =	Yes	
176	B0		Reserved (binary 0)		Bits 1-
177	B1		Reserved (binary 0)	Char(47)	
224	E0	Program data	Char(*)		
224	E0		Instruction stream component	Char(*)	
*	*		ODV component	Char(*)	
*	*		OES component	Char(*)	
*	*	BOM table	Char(*)		
*	*	Symbol table	Char(*)		
*	*	Object mapping table	Char(*) +		
*	*	—- End —-			

The first 4 bytes of the materialization template identify the total **number of bytes provided** in the template. This value is supplied as input to the instruction and is not modified. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization template are modified by the instruction to contain a value identifying the template size required to provide for the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified by the receiver. If the byte area identified by the receiver is greater than that required to contain the information

requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

The **existence attribute** indicates whether the program is *temporary* or *permanent*.

If the program has an associated space, then the **space attribute** is set to indicate either fixed- or variable-length; the initial value for the space is returned in the **initial value of space** field, and the **size of space** field is set to the current size value of the space. If the program has no associated space, the *size of space* field is set to a zero value, and the *space attribute* and *initial value of space* field values are meaningless.

If the program is addressed by a context, then the **context addressability** attribute is set to indicate this, and a system pointer to the addressing context is returned in the **context** field. If the program is not addressed by a context, then the *context addressability* attribute is set to indicate this, and binary 0's are returned in the *context* field.

If the program is a member of an access group, then the **access group** attribute is set to indicate this, and a system pointer to the access group is returned in the **access group** field. If the program is not a member of an access group, then the *access group* attribute is set to indicate this, and binary 0's are returned in the *access group* field.

The **automatically extend space** field controls whether the space is to be extended automatically by the machine or a *space addressing violation* (hex 0601) exception is to be signaled when a reference is made to an area beyond the allocated portion of the space. A value of binary 1 indicates the space will automatically be extended by an amount determined through internal machine algorithms. A value of binary 0 indicates the exception will result. Note that an attempt to reference an area beyond the maximum size that a space can be allocated, will always result in the signaling of the *space addressing violation* (hex 0601) exception independently of the setting of this attribute. A value of binary 1 is only valid when the *space attribute* has been specified as *variable length*.

Usage of the automatically extend space function is limited. Predictable results will occur only when you ensure that the automatic extension of a space will not happen in conjunction with modification of the space size by another thread. That is, you must ensure that when a thread is using the space in a manner that could cause it to be automatically extended, it is the sole thread which can cause the space size to be modified. Note that in addition to implicit modification through automatic extension, the space size can be explicitly modified through use of the Modify Space Attributes (MODS) instruction.

The **associated space hardware storage protection level** can be used to restrict access to the contents of the space by user state programs. It is possible to limit the access of the space by user state programs into 1 of three levels:

-
- *Reference only* (non-modifying storage references are allowed, modifying storing storage references yield an *object domain or hardware storage protection violation* (hex 4401) exception).
- *No storage references* (all storage references, modifying or non-modifying yield an *object domain or hardware storage protection violation* (hex 4401) exception).
- *Full access* (both modifying and non-modifying storage references are allowed).

The actual presentation of the *object domain or hardware storage protection violation* (hex 4401) exception is also dependent on the level of the physical hardware (namely, the CPU).

The **performance class** field provides information that allows the machine to more effectively manage the program by considering overall performance objectives of operations involving the program.

The primary associated space, if one is created, is always aligned on at least a 512 byte boundary if the *target version, release, and modification level* is V4R4 or greater. If the *target version, release, and modification level* is not V4R4 or greater, the primary associated space, if one is created, is always aligned on at least a 16-byte boundary.

If the **adopted user profile** attribute is *yes*, any reference to a system object from an invocation of this program uses the user profile of the owner of this program and other sources of authority to determine the authorization to system objects, privileged instructions, ownership rights, and all authorizations. If the **adopted user profile propagation** attribute is *yes*, then the authorities available from the adopted user profile are available to any further invocations while this program is invoked. If the *adopted user profile propagation* attribute is *no*, then the authorities available to the program's owning user profile are not available to further subinvocations and are available only to this invocation. These attributes do not affect the propagation of authority from higher existing invocations.

The **array constraint** field determines how array bounds should be checked at execution time. If *arrays are constrained*, execution time checks are made to verify that the array index is within the bounds of the array. If *arrays are unconstrained* and the *type of unconstrained array* is *not fully unconstrained*, the array references are assumed to be within the bounds of the array. If an array element reference is made outside the bounds of the array, unpredictable results may occur. If the *type of unconstrained array* is *fully unconstrained*, array references outside the bounds of the array will be made as if the elements existed. Array references of this type will signal the *space addressing violation* (hex 0601) exception if the element that is referenced is outside the allocated storage of the space containing the array. It is possible to change the type of constrainment used when referencing array elements by using the Override Program Attributes (OVRPGATR) instruction.

The **string constrainment** field determines how string limits should be checked at execution time. If *string constrainment* is *strings are not constrained*, the references are assumed to be within the defined bounds of the string. No execution time checks are performed to ensure this is the case. However, if the reference is outside the defined bounds, unpredictable results may occur. There may be significant savings in performance if *strings are not constrained* is specified. It is possible to change the type of constrainment used when substringing by using the Override Program Attributes (OVRPGATR) instruction.

Whenever a new invocation or activation is allocated, the automatic or static storage areas are initialized to bytes of binary 0's, respectively. The **static storage** and **automatic storage** program attributes control this default initialization. There is a significant performance advantage when these areas are not initialized by default. However, initial values specified for individual data objects are still set. The *automatic storage* and *static storage* will be allocated in single level store.

The **associated journal entry** field controls which program is associated with a journal entry. As a journal entry is made, a newest-to-oldest interrogation of the invocation stack is performed. The first program encountered that has the *associated journal entry* field set to *program name is recorded in journal entries* is associated with the journal entry by a record of the program name in the journal entry. If a program is encountered for which the *associated journal entry* field is set to *program name is not recorded in journal entry*, the program is ignored unless the program is on the top of the invocation stack. If the program is on the top of the invocation stack, it is associated with the journal entry by a record of the program name in the journal entry.

The **suppress decimal data exception** field controls whether or not errors detected in decimal data are to result in the signaling of the decimal data exception. When the decimal data exception *is not to be suppressed*, decimal values input to numeric operations are verified to contain valid decimal digit and sign codes with the *decimal data* (hex 0C02) exception being signaled as the result of detection of an invalid code. When the decimal data exception *is to be suppressed*, decimal values input to numeric operations are still verified to contain valid decimal digit and sign codes. However, detection of an invalid code results in the instruction interpreting an invalid digit as a zero and an invalid sign as positive rather than in signaling of the exception.

The control over suppressing of the decimal data exception applies to only a subset of the numeric operations which may be performed by the machine.

In general, when suppression of the decimal data exception is specified, the Computation and Branching instructions which support numeric source operands will suppress decimal data errors for ZONED source values. Additionally, a few instructions also suppress decimal data errors for packed source values. In all cases, the suppression is only supported when all operands are non-Data Pointer defined (early bound) source values.

Specifically, the following defines the exact list of instructions which support suppression of decimal data errors for certain operand types:

Packed Source Operands suppressed	Zoned Source Operands suppressed	Instruction Mnemonic	Note
yes	yes	ADDN	
yes	yes	CMPNV	
yes	yes	CVTDFP	Only supported when operand 3, the numeric view to be used for operand 2, is specified as a constant and no DP defined operands are specified.
yes	yes	CVTFDP	
yes	yes	CVTNC	Only supported when operand 3, the numeric view to be used for operand 1, is specified as a constant and no DP defined operands are specified.
yes	yes	CPYNV	Only supported when no DP defined operands are specified.
yes	yes	DIV	
yes	yes	DIVREM	
yes	yes	EDIT	Only supported when no DP defined operands are specified.
yes	yes	EXTRMAG	
yes	yes	MULT	
yes	yes	NEG	
yes	yes	REM	
yes	yes	SCALE	
yes	yes	SUBN	

When suppression of the decimal data exception is specified, the *decimal data* (hex 0C02) exception may still be signaled by the machine. That is, other instructions and instruction combinations not listed above are exposed to signaling the *decimal data* (hex 0C02) exception when invalid decimal data is encountered and they can't be counted on to fix up invalid decimal data.

Of particular note here is that specification of a Data Pointer as an input operand on any of the instructions in the above list negates the suppression support for the whole instruction. Thus even non-Data Pointer defined operands are then exposed to detection of the *decimal data* (hex 0C02) exception.

The **template extension existence** field indicates whether or not there is an extension to the program template.

The **suppress previously adopted user profiles** field is used to stop the backward search for adopted or propagated authorities. In other words, all programs previous to this program's invocation with adopted or propagated authorities are not considered as sources of authority.

The **code generation options** provide information that allows the machine to create a program based on the intended use of the program.

If the *performance optimization* field is binary 1, additional processing is performed which provides for creating a program that requires less processor resource for execution. This optimizes addressability used within the program. If this attribute is set to binary 0, normal optimization processing is performed.

The **space pointer machine objects** field controls whether space pointer machine objects are allowed in the ODV. If the *allow space pointer machine objects in ODV component* attribute is set to binary 1, additional processing is performed which allows for space pointer machine objects within the program. If this attribute is set to binary 0, space pointer machine objects are not allowed in the ODV component.

The **coincident operand overlap** field controls whether or not additional processing is performed during the encapsulation of certain computation and branching instructions which affects the processor resource required to execute these instructions. The effect of the option controls whether or not the encapsulation process for these instructions should assume that coincident operand overlap may occur between the source and receiver operands during execution of the instruction. This assumption applies to cases of nonidentical coincident operand overlap where the Create Program (CRTPG) instruction cannot determine if coincident operand overlap may occur during execution of the instruction. These instructions may produce invalid results if nonidentical coincident overlap occurs during execution, but the instruction was encapsulated with the assumption that it would not occur.

Specifying the *do not assume coincident operand overlap* attribute indicates that nonidentical coincident overlap will not occur during execution and therefore the receiver on an instruction may be used as a work area during operations performed to produce the final result. Using the receiver as a work area does not require the processor resource that would be required to move the final result from an internal work area to the receiver.

Specifying the *assume coincident operand overlap* attribute indicates that nonidentical coincident operand overlap may occur during execution and therefore the receiver on an instruction should not be used as a work area during operations that produce the final result. This can require more processor resource for instruction execution but it insures valid results if overlap occurs.

The following is a list of instructions that can be affected by the coincident operand overlap option during the encapsulation process:

- Add Logical Character
- Add Numeric
- And
- Compute Math Function Using One Input Value
- Concatenate
- Convert Character To Numeric
- Convert Decimal Form To Floating-Point
- Convert External Form To Numeric Value
- Convert Floating-Point To Decimal Form
- Convert Numeric To Character
- Copy Bytes Left Adjusted With Pad
- Copy Bytes Right Adjusted With Pad
- Divide
- Divide With Remainder
- Exclusive OR
- Multiply
- Or
- Remainder
- Scale
- Subtract Logical Character
- Subtract Numeric
- Trim Length

The **teraspace capable** option indicates whether or not the program produced should be enabled to use teraspace addresses.

The **executable part compression** field and **observation part compression** field indicate whether the executable, observation, or both parts of the program are to be compressed. For materialization, these fields indicate whether parts of the program object are currently compressed.

The **observation attributes** field specifies options that control the observability and debugability of the program.

The first six bits control the availability of information through the Materialize Program (MATPG) instruction. If a bit is a binary 1 then the corresponding data from the program template is available for materialization. If the program is created without the ability to materialize observability data then less storage may be needed to contain the program object.

The remaining two bits control whether certain performance measurements will be possible when the program is executing. **Prevent performance measurements on entry/exit** controls whether performance measurements can be made which encompass the duration of the execution of this program. **Prevent performance measurements on CALLX** controls whether performance measurements can be made which encompass the duration of a CALLX from this program to another program. If either bit is a binary 1 then the corresponding measurement is prevented. If the ability to make performance measurements is prevented then the program may execute more quickly.

The **size of static storage** field defines the total amount of static storage required for this program's static data. A value of 0 indicates that the amount of static storage required is calculated based upon the amount of static data specified for the program. A value greater than 0 specifies the amount of static storage required.

The **size of automatic storage** field defines the total amount of automatic storage required for this program's automatic data. A value of 0 indicates that the amount of automatic storage required is calculated based upon the amount of automatic data specified for the program. A value greater than 0 specifies the amount of automatic storage required.

The **number of instructions** fields (1 and 2) and **number of ODV entries** fields (1 and 2) is specified in different locations in the template depending on the version of the program template. Template version 0 limits the number of instructions to a maximum of 65,532 and the number of ODV entries to a maximum of 8,191. Programs that exceed one of these maximums cannot be created with template version 0. Template version 1 limits the number of instructions to a maximum of 65,532 and the number of ODV entries to a maximum of 65,526. Programs that exceed one of these maximums cannot be created with template version 1. All other values for the template version are reserved.

The **extended program attributes** allow for additional attributes of the program to be specified.

To **suppress binary size exceptions** indicates the *size* (hex 0C0A) exception will be suppressed when an overflow or underflow occurs on a computation and control instruction with a receiver that is a binary variable scalar. The receiver will contain the left-truncated result.

Create program for previous mandatory release indicates whether or not the program is created to run on the previous mandatory release

The **collect object usage data for programs** field is used to tell CALLX and XCTL instructions whether or not to collect object usage data for the program being called or transferred to.

The **scope of resources** field identifies the scope of program resources. The machine will set this field for the affected invocations, but the definition of those resources and the use of this field is determined by the MI user. If the *scope of resources* field is binary 0, then the resources will be scoped to the invocation of

this program. If this *scope of resources* field is binary 1, then the resources will be scoped to the previous invocation of this program. If the *scope of resources* is also binary 1 for that invocation, the resources will be scoped to the next previous invocation, and so on.

The **language version, release, and modification level** is used to limit which version, release, and modification level that this program is allowed to be moved back to. This attribute allows the compilers to specify the earliest release in which the necessary runtime environment exists for the program to execute. The program will not be allowed to be restored to a system running at an earlier release than the one identified. A zero value for this attribute means that no restriction is specified.

The **use new BOM table format** flag is used to indicate which format of the BOM table is used. Binary 0 indicates old format, binary 1 indicates new format. These formats are documented below in the *BOM table component*.

The **instruction stream component** consists of a 4-byte binary value that defines the total length of the instruction stream component and a variable-length array of 2-byte entries that defines the instruction stream. The 2-byte entries define instruction operation codes, instruction operation code extenders, or instruction operands.

See Operation Code Field for the format of the instructions. The instruction stream component is optional (that is, instructions need not be defined), and its absence is indicated by a value of 0 in the offset to instruction stream component entry. If the instruction stream is not present, an End instruction is assumed and, should the program be executed, an immediate Return External instruction results.

The **object definition vector** (ODV) component consists of a 4-byte binary value that defines the total length of the ODV and a variable-length vector of 4-byte entries. Each entry describes a program object either by a complete description or through an offset into the OES (object entry string) to a location that contains a description. If no program objects are defined, the ODV can be omitted, and its absence is noted with a value of 0 in the offset to ODV component entry. The ODV is required if the OES is present.

The **ODV entry string** (OES) consists of a 4-byte binary value that defines the total length of the OES and a series of variable-length entries that are used to complete an object description. Entries in the ODV contain offsets into the OES. The OES is optional, and its absence is indicated with a value of 0 in the offset to OES component entry.

The format of the ODT (object definition table) (ODV and OES) is defined in Program Object Specification.

The **BOM table component** can be used by compilers to relate high-level language statement numbers to instruction numbers.

The **BOM table** has the 2 formats depending on the *BOM table flags*. If the flag indicates to *use new BOM table format*, then the first bit of the *MI instruction number* is not a flag, so numbers up to 64k-1 can be used.

The **BOM table** has the following OLD format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	MI instruction number	UBin(2)	
0	0	Format		Bit 0

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = High level statement number is in character format 1 = High level statement number is in numeric format	
0	0		MI instruction number	Bits 1-15
2	2	High level statement number If this is in *character format*, then the length of it is contained in the header in the *length of breakpoint offset mapping table entry*.	Char(*) or Bin(2)	
*	*	—- End —-		

The **BOM table** has the following NEW format:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	MI instruction number	UBin(2)	
2	2	Flag Byte	Char(1)	
2	2		Format	Bit 0
			0 = High level statement number is in character format 1 = High level statement number is in numeric format	
2	2		Reserved	Bits 1-7
3	3	High level statement number If this is in *character format*, then the length of it is contained in the header in the *length of breakpoint offset mapping table entry*.	Char(*) or Bin(2)	
*	*	—- End —-		

The **symbol table component** can be used by compilers to relate high-level language names to ODT numbers.

The **symbol table** has the following format:

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Hashing table structure	Char(*)	
0	0		Number of hash buckets	Bin(4)
4	4		Hash bucket Each hash bucket contains an offset to the first symbol table base segment entry of the chain. This offset is from the beginning of the symbol table. The end of the chain has a -1 value. Maximum of 1000 hash buckets.	[*] Bin(4)
*	*	Symbol table base segment	Char(*)	
*	*		Offset to next entry from beginning of the table The end of the chain has a -1 value.	UBin(4)
*	*		ODT or MI number	Bin(2)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
*	*		Indicators	Char(1)
*	*		Instruction or ODT number	Bit 0
			0 = MI instruction number	
			1 = ODT number	
*	*		Symbol origin	Bit 1
			0 = Compiler generated	
			1 = Source program	
*	*		Array specification	Bit 2
			0 = Row major	
			1 = Column major	
*	*		Format segment present	Bit 3
			0 = No	
			1 = Yes	
*	*		Array segment present	Bit 4
			0 = No	
			1 = Yes	
*	*		Extension segment present	Bit 5
			0 = No	
			1 = Yes	
*	*		Reserved (binary 0)	Bits 6-
*	*		Length of symbol	Char(1)
*	*		Symbol	Char(*)
*	*	—- End —-		

Other segments are only present if the bit in the symbol table base segment is on.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Symbol table format segment	Char(20)	
0	0		Format program name	Char(10)
10	A		Format code	Char(4)
14	E		Locator variable ODT#	Bin(2)
16	10		Descriptor variable ODT#	Bin(2)
18	12		Reserved (binary 0)	Char(2)
20	14	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Symbol table array segment	Char(*)	
0	0		Number of array dimensions	Bin(2)
2	2		Indexes - 1 per array dimension	[*] Char(8)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
2	2		Lower index	Bin(4)
6	6		Upper index	Bin(4)
*	*	—- End —-		

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Symbol table extended segment	Char(26)	
0	0		Extended segment length	Bin(2)
2	2		Structure level	Char(2
4	4		Data representation	Char(1
			Hex 00 = See ODT	
			Hex 01 = Binary	
			Hex 02 = Zoned	
			Hex 03 = Bit string	
5	5		Number of total digits	Bin(2)
7	7		Number of fractional digits	Bin(2)
9	9		Sign of number	Char(1
			Hex 00 = Leading embedded	
			Hex 01 = Leading separated	
			Hex 02 = Trailing separate	
10	A		Offset to base segment entry of parent The end of the chain has a -1 value.	Bin(4)
14	E		Offset to base segment entry of synonym The end of the chain has a -1 value.	Bin(4)
18	12		Indicators	Char(1
18	12		Object is a HLL pointer	
18	12		Array segment is in multi-dimensioned array format	
18	12		Reserved (binary 0)	
19	13		Reserved (binary 0)	Char(7
26	1A	—- End —-		

Hashing is done by exclusively Or'ing the first 4 characters of the symbol name with the second 4 characters of the symbol name. The result is then divided by the *number of hash buckets*. If the result is negative or 0, the *number of hash buckets* is added to the result. The result is then used as an index to the *hash bucket.*

Format segment is used by certain compilers to specify a *format program name* to be used when formatting this variable.

The **offset to the OMT component** field specifies the location of the OMT component in the materialized program template. The OMT consists of a variable-length vector of 6-byte entries. The number of entries is identical to the number of ODV entries because there is one OMT entry for each ODV entry. The OMT entries correspond one for one with the ODV entries; each OMT entry gives a location mapping for the object defined by its associated ODV entry.

The following describes the formats for an OMT entry:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	OMT entry	Char(6)
0	0	Addressability type	Char(1)

Hex 00=

Base addressability is from the start of the static storage

Hex 01=

Base addressability is from the start of the automatic storage area

Hex 02=

Base addressability is from the start of the storage area addressed by a space pointer

Hex 03=

Base addressability is from the start of the storage area of a parameter

Hex 04=

Base addressability is from the start of the storage area addressed by the space pointer found in the process communication object attribute of the process associated with the thread executing the program

Hex FF=

Base addressability not provided. The object is contained in machine storage areas to which addressability cannot be given, or a parameter has addressability to an object that is in the storage of another program

1	1	Offset from base	Char(3)

Offset Dec	Hex	Field Name	Data Type and Length
			For types hex 00, hex 01, hex 02, hex 03, and hex 04, this is a 3-byte logical binary value representing the offset to the object from the base addressability. For type hex FF, the value is binary 0.
			4 4 Base addressability Char(2)

For types hex 02 and hex 03, this is a 2-byte binary field containing the number of the OMT entry for the space pointer or a parameter that provides base addressability for this object. For types hex 00, hex 01, hex 04 and hex FF, the value is binary 0.

6
6
—- End —-

Warning: Temporary Level 3 Header

Authorization Required

•

- Retrieve
  –
  – Operand 2
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Operand 2
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220A Program Not Eligible for Operation

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Footnotes:

[1] The previous mandatory release is release N-1, mod level zero when release N is the current release.. (For version 4, release 5.0, the previous mandatory release is version 4, release 4.0.).

Materialize Program Name (MATPGMNM)

Bound program access
Built-in number for MATPGMNM is 473. MATPGMNM (receiver_template : address)

Description: This instruction will return the program and context names in operand 1 of the bound program, bound service program, or Java program associated with the currently executing procedure. Operand 1 must be aligned on a 16-byte boundary; otherwise a *boundary alignment* (hex 0602) exception is signaled. The format of the receiver template is:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided by the user	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Format	Bin(4)	

0 = Program in context format

12
C
Reserved
Char(4)
16
10
Formatted data
Char(*)
*
*
—- End —-

The **formatted data** field is structured based on the value specified in the *format* field. For *format* = 0, the *formatted data* field has the following structure.

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
16	10	Bound program context object type	Char(1)
17	11	Bound program context object subtype	Char(1)
18	12	Context name	Char(30)
48	30	Bound program object type	Char(1)
49	31	Bound program object subtype	Char(1)
50	32	Program name	Char(30)
80	50	—- End —-	

The first 4 bytes that are materialized identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 16 causes a *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes that are materialized identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the materialization length and boundary alignment exceptions described previously) are signaled in the event that the receiver contains insufficient area for the materialization. For *format* 0, if any field cannot be completely materialized, blanks will be returned in the partial field.

If the program is logically destroyed before this instruction is executed, the *context name* will be returned as all blank characters, and the *context object type* and *context object subtype* fields will be returned as zeroes.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Queue Attributes (MATQAT)

Op Code (Hex)	Operand 1	Operand 2
0336	Receiver	Queue

Operand 1: Space pointer.

Operand 2: System pointer.

Bound program access
Built-in number for MATQAT is 44. MATQAT (receiver : address queue : address of system pointer)

Description: The attributes of the *queue* specified by operand 2 are materialized into the *receiver* specified by operand 1. The *receiver* must be aligned on a 16-byte multiple. The format of the materialized queue attributes is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Object identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Object creation options	Char(4)	
40	28		Existence attributes	Bit 0
			0 = Temporary	
			1 = Permanent	
40	28		Space attribute	Bit 1
			0 = Fixed-length	
			1 = Variable-length	
40	28		Initial context	Bit 2
			0 = Addressability not in context	
			1 = Addressability in context	
40	28		Access group	Bit 3
			0 = Not a member of access group	
			1 = Member of access group	
40	28		Reserved (binary 0)	Bits 4-12
40	28		Initialize space	Bit 13
40	28		Reserved (binary 0)	Bits 14-31
44	2C	Reserved (binary 0)	Char(4)	
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
53	35		Space alignment	Bit 0

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space.	
			1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space.	
			Ignore the value of this field when the *machine chooses space alignment* field has a value of binary 1.	
53	35		Reserved (binary 0)	Bits 1-2
53	35		Machine chooses space alignment	Bit 3
			0 = The space alignment indicated by the *space alignment* field is in effect.	
			1 = The machine chose the space alignment most beneficial to performance, which may have reduced maximum space capacity. The alignment chosen is a multiple of 512. Ignore the value of the *space alignment* field.	
53	35		Reserved (binary 0)	Bit 4
53	35		Main storage pool selection	Bit 5
			0 = Process default main storage pool is used for object.	
			1 = Machine default main storage pool is used for object.	
53	35		Reserved (binary 0)	Bit 6
53	35		Block transfer on implicit access state modification	Bit 7
			0 = Transfer the minimum storage transfer size for this object.	
			1 = Transfer the machine default storage transfer size for this object.	
53	35		Reserved (binary 0)	Bits 8-31
57	39	Reserved (binary 0)	Char(7)	
64	40	Context	System pointer	
80	50	Access group	System pointer	
96	60	Queue attributes	Char(1)	
96	60		Message content	Bit 0
			0 = Contains scalar data only	
			1 = Contains pointers and scalar data	
96	60		Queue type	Bits 1-2
			00 = Keyed	
			01 = Last in, first out (LIFO)	
			10 = First in, first out (FIFO)	
96	60		Queue overflow action	Bit 3

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = Signal exception	
			1 = Extend queue	
96	60		Choose maximum number of extends	Bit 4
			0 = Machine chooses maximum number of extends	
			1 = User specifies maximum number of extends	
96	60		Reclaim storage	Bit 5
			0 = Do not reclaim queue storage	
			1 = Reclaim storage when messages enqueued is zero	
96	60		Reserved (binary 0)	Bits 6-7
97	61	Current maximum number of messages	Bin(4)	
101	65	Current number of messages enqueued	Bin(4)	
105	69	Extension value	Bin(4)	
109	6D	Key length	Bin(2)	
111	6F	Maximum size of message to be enqueued	Bin(4)	
115	73	Reserved (binary 0)	Char(1)	
116	74	Maximum number of extends	Bin(4)	
120	78	Current number of extends	Bin(4)	
124	7C	Initial number of messages	Bin(4)	
128	80	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception described previously) are signaled when the receiver contains insufficient area for the materialization.

The following fields in the template are returned. No queue attributes are modified by the MATQAT instruction.

The **object identification** specifies the symbolic name that identifies the queue within the machine. An **object type** of hex 0A is implicitly supplied by the machine. The *object identification* is used to identify the object on materialize instructions as well as to locate the object in a context that addresses the object.

If the created object is permanent, it is owned by the user profile governing thread execution when the queue was created. The owning user profile is implicitly assigned all private authority states for the object. The storage occupied by the created object is charged to this owning user profile. If the created object is temporary, no owning user profile exists, and all authority states are assigned as public. Storage occupied by the created object is charged to the creating process.

The **existence attribute** specifies whether the queue is to be created as *temporary* or *permanent*. A *temporary* queue, if not explicitly destroyed by the user, is implicitly destroyed by the machine when machine processing is terminated.

If a space is associated with the queue, the space may be *fixed* or *variable* in size, as specified by the **space attribute**. The current allocation is as specified in the **size of space** field. The machine allocates a space of at least the size specified at queue creation; the actual size allocated depends on an algorithm defined by a specific implementation.

If the **initial context** attribute field indicates that *addressability is inserted in a context*, the **context** field contains a system pointer that identifies the context where addressability to the queue is placed.

If the **access group** creation attribute field indicates *member of access group*, the **access group** field contains a system pointer that identifies the access group in which the object was created. Only temporary queues may be created in an access group.

The **initialize space** creation option controls whether or not the space is to be initialized. When *initialize* is specified, each byte of the space is initialized to the value specified by the **initial value of space** field. Additionally, when the space is extended in size, this byte value is also used to initialize the new allocation. When *do not initialize* is specified, the *initial value of space* field is ignored and the initial value of the bytes of the space are unpredictable.

When *do not initialize* is specified for a space, internal machine algorithms do ensure that any storage resources last used for allocations to another object which are reused to satisfy allocations for the space are reset to a machine default value to avoid possible access of data which may have been stored in the other object. To the contrary, reuse of storage areas previously used by the space object are not reset, thereby exposing subsequent reallocations of those storage areas within the space to access of the data which was previously stored within them.

The **message content** attribute specifies whether the messages to be enqueued will *contain pointers and scalar data*, or *scalar data only*. If the messages are to contain pointers, the message text operand on Enqueue and Dequeue instructions must be aligned on 16-byte boundaries.

The **queue type** attribute establishes the basic sequence in which messages are dequeued from the queue.

The **current number of messages enqueued** field contains the number of messages currently enqueued on the queue.

The **queue overflow action** field establishes the machine action when the number of messages resident on the queue (enqueued and not yet dequeued) exceeds the current maximum capacity of the queue. This value is initially established by the value specified in the **maximum number of messages** field. The *queue full* (hex 2602) exception is signalled when the number of resident messages exceeds this field unless the extend queue option is specified. When the *extend queue* option is specified for the *queue overflow action* field, the value of the *current maximum number of messages* field is increased by the amount specified by the **extension value** field each time the number of enqueued messages exceeds the value of the *current maximum number of messages* field. When the *extend queue* option is specified for the *queue overflow action* field, the *extension value* field contains a value greater than 0. If the *signal exception* option is specified, the *extension value* field is ignored and the *current maximum number of messages* field contains a value greater than zero.

The **choose maximum number of extends** field allows the user to overide the value for the maximum number of extends to the queue which would otherwise be chosen by the machine. If this field specifies *machine chooses maximum number of extends*, then the number of extends will be chosen such that the maximum number of messages for the queue will never be greater than what can be completely materialized into 16MB or require overall object size greater than 32MB. The overall object size depends upon the amount of storage needed for queue definition plus entries enqueued to queue and excludes the size of the associated space, if any. If this field specifies *user specifies maximum number of extends*, the queue will be extended by the number of messages specified by the *extension value* field until the number of extends reaches the value returned by the **maximum number of extends** field is reached. The **current number of extends** field specifies the number of times the queue has currently been extended.

The **reclaim storage** field specifies whether storage reclaim will be attempted when the number of currently enqueued messages on the queue reaches zero. If this field specifies *reclaim storage when messages enqueued is zero* then the size of the queue will be reduced to the number of messages specified by the **initial number of messages** field when the queue was created. The *current number of extends* field is reset to zero after the queue is reclaimed. If this field specifies *do not reclaim queue storage* then no action is taken.

The **key length** field establishes the size of the queue's key. The key can contain pointers, but the pointers are considered to be scalar data when they are placed on the queue by an Enqueue instruction. If the *queue type* field specifies LIFO or FIFO, the *key length* can be equal to or greater than 0; however, the queue is not treated as a keyed queue.

The size of all messages to be enqueued is established by the **maximum size of messages to be enqueued** field. The Enqueue instruction may specify a size (in the message prefix) that is greater than this value, but the message is truncated to this length. The *maximum size of messages to be enqueued* field has a value of 0 or greater, up to a maximum value of 64 K bytes. The maximum size of a queue, excluding its associated space, cannot exceed 2 gigabytes. This value includes machine overhead associated with the queue.

Limitations (Subject to Change): The following are limits that apply to the functions performed by this instruction. These limits may change on different implementations of the machine.

The size of the object specific portion of this object is limited to a maximum of 2 gigabytes. This size is dependent upon the amount of storage needed for the queue definition plus entries enqueued to queue and excludes the size of the associated space, if any.

The size of the associated space for this object is limited to a maximum of 16MB-32 bytes if the machine does not choose the space alignment and 0 is specified for the *space alignment* field. The size of the associated space for this object is limited to a maximum of 16MB-512 bytes if the machine does not choose the space alignment and 1 is specified for the *space alignment* field. The maximum size of an associated space for this object if the machine choose the space alignment is returned by option Hex 0003 of MATMDATA.

Warning: Temporary Level 3 Header

Authorization Required

•

- Operational
  - –
  - – Operand 2
- Execute
  - –
  - – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  - –
  - – Operand 2
  - – Contexts referenced for address resolution

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

0A Authorization

- 0A01 Unauthorized for Operation

10 Damage Encountered

- 1004 System Object Damage State
- 1005 Authority Verification Terminated Due to Damaged Object
- 1044 Partial System Object Damage

1A Lock State

- 1A01 Invalid Lock State

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2207 Authority Verification Terminated Due to Destroyed Object
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid
- 2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Queue Messages (MATQMSG)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
033B	Receiver	Queue	Message selection template

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: Character(16) scalar.

Bound program access

```
Built-in number for MATQMSG is 45.
MATQMSG (
         receiver                    : address
         queue                       : address of system pointer
         message_selection_template  : address
)
```

Description: Materialize selected messages on a queue. One or more messages on the *queue* specified by operand 2 are selected according to information provided in operand 3 and materialized into operand 1. The number of messages materialized and the amount of key and message text data materialized for each message is governed by the *message selection template*.

Note that the list of messages on a queue is a dynamic attribute and may be changing on a continual basis. The materialization of messages provided by this instruction is just a picture of the status of the queue at the point of interrogation by this instruction. As such, the actual status of the queue may differ from that described in the materialization when subsequent instructions use the information in the template as a basis for operations against the queue.

Operand 1 specifies a space that is to receive the materialized attribute values.

Operand 2 is a system pointer identifying the *queue* from which the messages are to be materialized.

Operand 3 is a character (16) scalar specifying which messages are to be materialized.

The operand 1 space pointer must address a 16-byte boundary. The materialization template has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Materialization size specification	Char(8)		
0	0		Number of bytes provided for materialization	Bin(4)	
4	4		Number of bytes available for materialization	Bin(4)	
8	8	Materialization data	Char(4)		
8	8		Count of messages materialized	Bin(4)	
12	C	Queue data	Char(12)		
12	C		Count of messages on the queue	Bin(4)	
16	10		Maximum message size	Bin(4)	
20	14		Key size	Bin(4)	
24	18	Reserved	Char(8)		
32	20	Message data	[*] Char(*)		
		(repeated for each message)			
32	20		Message attributes	Char(16)	
32	20		Message enqueue time		Char(8)
40	28		Message length		Bin(4)
44	2C		Reserved		Char(4)
48	30		Message key	Char(*)	
*	*		Message text	Char(*)	
*	*	—- End —-			

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception described previously.

The **maximum message size** and **key size** are values specified when the queue was created. If the queue is not a keyed queue, the value materialized for the key size is zero.

The length of the *message key* and *message text* fields is determined by values supplied in operand 3, message selection data. If the length supplied in operand 3 exceeds the actual data length, the remaining space will be padded with binary zeros.

The *message selection template* identified by operand 3 must be at least 16 bytes and must be on a 16-byte boundary. The format of the *message selection template* is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Message selection	Char(2)	
0	0		Type	Bits 0-3

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0001 = All messages	
			0010 = First	
			0100 = Last	
			1000 = Keyed	
			All other values are reserved	
0	0		Key relationship (if needed)	Bits 4-7
			0010 = Greater than	
			0100 = Less than	
			0110 = Not equal	
			1000 = Equal	
			1010 = Greater than or equal	
			1100 = Less than or equal	
			All other values are reserved	
0	0		Reserved	Bits 8-15
2	2	Lengths	Char(8)	
2	2		Number of key bytes to materialize	Bin(4)
6	6		Number of message text bytes to materialize	Bin(4)
10	A	Reserved	Char(6)	
16	10	Key (if needed)	Char(*)	
*	*	—- End —-		

The **message selection type** must not specify *keyed* if the queue was not created as a keyed queue.

Both of the fields specified under **lengths** must be zero or an integer multiple of 16. The maximum value allowed for the key length is 256. The maximum value allowed for the message text is 65,536.

Warning: Temporary Level 3 Header

Authorization Required

•

• Retrieve

  –

  – Operand 2

• Execute

  –

  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialization

  –

  – Operand 2

  – Contexts referenced for address resolution

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

0A Authorization

- 0A01 Unauthorized for Operation

10 Damage Encountered

- 1004 System Object Damage State
- 1005 Authority Verification Terminated Due to Damaged Object
- 1044 Partial System Object Damage

1A Lock State

- 1A01 Invalid Lock State

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2207 Authority Verification Terminated Due to Destroyed Object
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid
- 2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Resource Management Data (MATRMD)

Op Code (Hex)	Operand 1	Operand 2
0352	Receiver	Control data

Operand 1: Space pointer.

Operand 2: Character(8) scalar.

Bound program access

```
Built-in number for MATRMD is 69.
MATRMD (
        receiver       : address
        control_data   : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The data items requested by operand 2 are materialized into the receiving object specified by operand 1. Operand 2 is an 8-byte character scalar. The first byte identifies the generic type of information being materialized, and the remaining 7 bytes further qualify the information desired. Unless otherwise stated, the data items requested for operand 2 are for the current partition.

Operand 1 contains the materialization and has the following format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Time of day	Char(8)	
16	10	Resource management data	Char(*)	
*	*	—- End —-		

The remainder of the materialization depends on operand 2 and on the machine implementation. The following values are allowed for operand 2:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Selection option	Char(1)

Offset Dec	Offset Hex	Field Name	Data Type and Length
		Hex 01 = Materialize original processor utilization data (option hex 26 is preferred) (See ″Original Processor Utilization (Hex 01)″ (page 836))	
		Hex 03 = Materialize storage management counters (See ″Storage Management Counters (Hex 03)″ (page 838))	
		Hex 04 = Materialize storage transient pool information (See ″Storage Transient Pool Information (Hex 04)″ (page 839))	
		Hex 08 = Materialize machine address threshold data (See ″Machine Address Threshold Data (Hex 08)″ (page 839))	
		Hex 09 = Materialize main storage pool information (See ″Main Storage Pool Information (Hex 09)″ (page 840))	
		Hex 0A = Materialize multiprogramming level (MPL) control information with 2-byte counts (option hex 16 is preferred) (See ″MPL Control Data with 2-byte counts (Hex 0A)″ (page 841))	
		Hex 0C = Materialize machine reserved storage pool information (See ″Machine Reserved Storage Pool Information (Hex 0C)″ (page 843))	
		Hex 11 = (Ignored) (See ″User storage area 1 - OBSOLETE (Hex 11)″ (page 843))	
		Hex 12 = Materialize auxiliary storage information for on-line ASPs. (See ″Auxiliary Storage Information (Hex 12)″ (page 843))	
		Hex 13 = Materialize original multiprocessor utilizations (option hex 28 is preferred) (See ″Original Multiprocessor utilizations (Hex 13)″ (page 859))	
		Hex 14 = Materialize storage pool tuning (See ″Storage pool tuning (Hex 14)″ (page 861))	
		Hex 15 = Materialize delay cost scheduling information (See ″Delay cost scheduling information (Hex 15)″ (page 865))	
		Hex 16 = Materialize MPL control information (4-byte counts) (See ″MPL Control Data (Hex 16)″ (page 865))	
		Hex 17 = Materialize allocation and de-allocation counts per task and thread (See ″Allocation and De-allocation counts per task and thread (Hex 17)″ (page 867))	
		Hex 18 = Materialize processor multi-tasking mode (See ″Processor Multi-tasking mode (hex 18)″	

Offset Dec	Offset Hex	Field Name	Data Type and Length
		Hex 20 = Materialize auxiliary storage information including varied-off independent ASPs (See "Auxiliary Storage information including offline Independent ASPs (Hex 20)" (page 883))	
		Hex 22 = Materialize auxiliary storage pool information including varied-off independent ASPs (See "Auxiliary Storage Pool Information including offline Independent ASPs (Hex 22)" (page 899))	
		Hex 23 = Materialize ASP group information (See "Auxiliary Storage Pool Group Information (Hex 23)" (page 901))	
		Hex 24 = Materialize dynamic thread resources affinity adjustment (page "Dynamic Thread Resources Affinity Adjustment (Hex 24)" (page 903))	
		Hex 25 = Materialize ASP space information (See "Auxiliary Storage Pool Space Information (Hex 25)" (page 904))	
		Hex 26 = Materialize processor utilization data (See "Processor Utilization Data (Hex 26)" (page 905))	
		Hex 27 = Materialize shared processor pool information (See "Shared Processor Pool Information (Hex 27)" (page 907))	
		Hex 28 = Materialize multiprocessor utilizations (See "Multiprocessor utilizations (Hex 28)" (page 908))	
		Hex 29 = Materialize machine resource portions (page "Materialize machine resource portions (Hex 29)" (page 909))	
		Hex 2A = Materialize interrupt polling control (page "Materialize interrupt polling control (Hex 2A)" (page 909))	
1	1	Reserved (binary 0)	Char(7)
8	8	—- End —-	

The following defines the formats and values associated with each of the above materializations of resource management data.

Original Processor Utilization (Hex 01):

Note: Option hex 26 is the preferred method of materializing processor utilization data.

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Processor time since IPL (initial program load)	Char(8)
24	18	Secondary workload processor time since IPL	Char(8)
32	20	Database processor time since IPL	Char(8)
40	28	Database threshold	UBin(2)
42	2A	Database limit	UBin(2)
44	2C	Reserved (binary 0)	UBin(4)
48	30	Interactive processor time since IPL	Char(8)
56	38	Interactive threshold	UBin(2)
58	3A	Interactive limit	UBin(2)
60	3C	Reserved (binary 0)	UBin(4)
64	40	—- End —-	

Processor time since IPL is the total amount of processor time used, both by threads and internal machine functions, since IPL. The significance of bits within the field is the same as that defined for the time-of-day clock. On a machine with more than one active virtual processor, the value returned will be the average of the processor time used since IPL by all virtual processors.

Note: For the definition of virtual processor, see MATMATR option hex 01E0.

Secondary workload processor time since IPL is the total processor time, used for workloads that can not fully exploit dedicated server resources, since IPL. If a system is not a dedicated server, a value of hex 0000000000000000 is returned. The significance of bits within this field is the same as that defined for the time-of-day clock. On a machine with more than one active virtual processor, the value returned will be the average of the processor time used since IPL by all virtual processors.

Database processor time since IPL is the total processor time, used performing database processing, since IPL. If the system does not support this metric, a value of hex 0000000000000000 is returned. If the system does support this and needs to return a value of 0, a value of hex 0000000000001000 is returned. For all other cases, the significance of bits within this field is the same as that defined for the time-of-day clock. On a machine with more than one active virtual processor, the value returned will be the average of the processor time used since IPL by all virtual processors.

Database threshold is the highest level of database processor utilization which can be sustained without causing a disproportionate increase in system overhead. The value returned is the fraction of processor capacity, expressed in tenths of a percent. For example, a value of 237 means that the threshold is 23.7%. On a machine with no limit on database utilization, the value returned will be 1000 (100%).

Database limit is the maximum sustainable level of database processor utilization. The value returned is the fraction of processor capacity, expressed in tenths of a percent. For example, a value of 275 means that the limit is 27.5%. On a machine with no limit on database utilization, the value returned will be 1000 (100%).

Interactive processor time since IPL is the total processor time, used by interactive processes, since IPL. If the system does not support this metric, a value of hex 0000000000000000 is returned. If the system does support this and needs to return a value of 0, a value of hex 0000000000001000 is returned. For all other cases, the significance of bits within this field is the same as that defined for the time-of-day clock. On a machine with more than one active virtual processor, the value returned will be the average of the processor time used since IPL by all virtual processors. An interactive process is any process doing 5250 display device I/O. For additional information on interactive processes, see manual SC41-0607 iSeries(TM) Performance Capabilities Reference manual which is available in the iSeries Information Center.

Interactive threshold is the highest level of interactive processor utilization which can be sustained without causing a disproportionate increase in system overhead. The value returned is the fraction of processor capacity, expressed in tenths of a percent. For example, a value of 237 means that the threshold is 23.7%. On a machine with no limit on interactive utilization, the value returned will be 1000 (100%).

Interactive limit is the maximum sustainable level of interactive processor utilization. The machine determines the *interactive limit* based on the *interactive feature*. The value returned is the fraction of processor capacity, expressed in tenths of a percent. For example, a value of 275 means that the limit is 27.5%. On a machine with no limit on interactive utilization, the value returned will be 1000 (100%).

In a partition sharing physical processors, *processor time since IPL*, *secondary workload processor time since IPL*, *database processor time since IPL*, and *interactive processor time since IPL* are scaled appropriately so that the CPU utilization calculations can be done as if the partition was using dedicated processors.

Storage Management Counters (Hex 03):

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
16	10	Access pending	Bin(2)
18	12	Storage pool delays	Bin(2)
20	14	Directory look-up operations	Bin(4)
24	18	Directory page faults	Bin(4)
28	1C	Access group member page faults	Bin(4)
32	20	Microcode page faults	Bin(4)
36	24	Microtask read operations	Bin(4)
40	28	Microtask write operations	Bin(4)
44	2C	Reserved	Bin(4)
48	30	—- End —-	

Access pending is a count of the number of times that a paging request must wait for the completion of a different request for the same page.

Storage pool delays is a count of the number of times that threads have been momentarily delayed by the unavailability of a main storage frame in the proper pool.

Directory look-up operations is a count of the number of times that auxiliary storage directories were interrogated, exclusive of storage allocation or de-allocation.

Directory page faults is a count of the number of times that a page of the auxiliary storage directory was transferred to main storage, to perform either a look-up or an allocation operation.

Access group member page faults is a count of the number of times that a page of an object contained in an access group was transferred to main storage.

Microcode page faults is a count of the number of times a page of LIC was transferred to main storage.

Microtask read operations is a count of the number of transfers of one or more pages of data from auxiliary main storage on behalf of a task rather than a thread.

Microtask write operations is a count of the number of transfers of one or more pages of data from main storage to auxiliary storage on behalf of a task, rather than a thread.

Storage Transient Pool Information (Hex 04):

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Storage pool to be used for the transient pool	Bin(2)
18	12	—- End —-	

The pool number materialized is the number of the main storage pool, which is being used as the transient storage pool. A value of 0 indicates that the transient pool attribute is being ignored.

Machine Address Threshold Data (Hex 08):

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Total permanent addresses possible	Char(8)
24	18	Total temporary addresses possible	Char(8)
32	20	Permanent addresses remaining	Char(8)
40	28	Temporary addresses remaining	Char(8)
48	30	Permanent addresses remaining threshold	Char(8)
56	38	Temporary addresses remaining threshold	Char(8)
64	40	Total permanent 4GB addresses possible	Char(8)
72	48	Total permanent 256MB addresses possible	Char(8)
80	50	Total temporary 4GB addresses possible	Char(8)
88	58	Total temporary 256MB addresses possible	Char(8)
96	60	Permanent 4GB addresses remaining	Char(8)
104	68	Permanent 256MB addresses remaining	Char(8)
112	70	Temporary 4GB addresses remaining	Char(8)
120	78	Temporary 256MB addresses remaining	Char(8)
128	80	Permanent 4GB addresses remaining threshold	Char(8)
136	88	Permanent 256MB addresses remaining threshold	Char(8)
144	90	Temporary 4GB addresses remaining threshold	Char(8)
152	98	Temporary 256MB addresses remaining threshold	Char(8)
160	A0	—- End —-	

Total permanent addresses possible is the maximum number of permanent addresses for the machine.

Total temporary addresses possible is the maximum number of temporary addresses for the machine.

Permanent addresses remaining is the number of permanent addresses that can still be created.

Temporary addresses remaining is the number of temporary addresses that can still be created.

Permanent addresses remaining threshold is a number that, when it exceeds the number of permanent addresses remaining, causes an event to be signaled.

Temporary addresses remaining threshold is a number that, when it exceeds the number of temporary addresses remaining, causes an event to be signaled.

Total permanent 4GB addresses possible is the maximum number of permanent 4GB addresses for the machine.

Total permanent 256MB addresses possible is the maximum number of permanent 256MB addresses for the machine.

Total temporary 4GB addresses possible is the maximum number of temporary 4GB addresses for the machine.

Total temporary 256MB addresses possible is the maximum number of temporary 256MB addresses for the machine.

Permanent 4GB addresses remaining is the number of permanent 4GB addresses that can still be created.

Permanent 256MB addresses remaining is the number of permanent 256MB addresses that can still be created.

Temporary 4GB addresses remaining is the number of temporary 4GB addresses that can still be created.

Temporary 256MB addresses remaining is the number of temporary 256MB addresses that can still be created.

Permanent 4GB addresses remaining threshold is a number that, when it exceeds the number of permanent 4GB addresses remaining, causes an event to be signaled if the *suppress 4GB permanent address usage event flag* is set to *no*.

Permanent 256MB addresses remaining threshold is a number that, when it exceeds the number of permanent 256MB addresses remaining, causes an event to be signaled if the *suppress 256 MG permanent address usage event flag* is set to *no*.

Temporary 4GB addresses remaining threshold is a number that, when it exceeds the number of temporary 4GB addresses remaining, causes an event to be signaled if the *suppress 4GB temporary address usage event flag* is set to *no*.

Temporary 256MB addresses remaining threshold is a number that, when it exceeds the number of temporary 256MB addresses remaining, causes an event to be signaled if the *suppress 256MB temporary address usage event flag* is set to *no*.

Main Storage Pool Information (Hex 09):

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
16	10	Machine minimum transfer size	Bin(2)	
18	12	Maximum number of pools	Bin(2)	
20	14	Current number of pools	Bin(2)	
22	16	Main storage size	Bin(4)	
26	1A	Reserved (binary 0)	Char(2)	
28	1C	Pool 1 minimum size	Bin(4)	
32	20	Individual main storage pool information	[*] Char(32)	
		(repeated once for each pool, up to the current number of pools)		
32	20	Pool size	Bin(4)	
36	24	Pool maintenance	Bin(4)	
40	28	Thread interruptions (data base)	Bin(4)	
44	2C	Thread interruptions (nondata base)	Bin(4)	
48	30	Data transferred to pool (data base)	Bin(4)	
52	34	Data transferred to pool (nondata base)	Bin(4)	
56	38	Amount of pool not assigned to virtual addresses	Bin(4)	
60	3C	Reserved (binary 0)	Char(4)	
*	*	—- End —-		

Machine minimum transfer size is the smallest number of bytes that may be transferred as a block to or from main storage.

Maximum number of pools is the maximum number of storage pools into which main storage may be partitioned. These pools will be assigned the logical identification beginning with 1 and continuing to the *maximum number of pools*.

Current number of pools is a user-specified value for the number of storage pools the user wishes to utilize. These are assumed to be numbered from 1 to the number specified. This number is fixed by the machine to be equal to the maximum number of pools.

Main storage size is the amount of main storage, in units equal to the *machine minimum transfer size*, which may be apportioned among main storage pools.

Pool 1 minimum size is the amount of main storage, in units equal to the *machine minimum transfer size*, which must remain in pool 1. This amount is machine and configuration dependent.

Individual main storage pool information is data in an array that is associated with a main storage pool by virtue of its ordinal position within the array. In the descriptions below, data base refers to all other data, including internal machine fields. *Pool size, pool maintenance, amount of pool not assigned to virtual addresses* and *data transferred information* is expressed in units equal to the *machine minimum transfer size* described above.

Pool size is the amount of main storage assigned to the pool.

Pool maintenance is the amount of data written from a pool to secondary storage by the machine to satisfy demand for resources from the pool. It does not represent total transfers from the pool to secondary storage, but rather is an indication of machine overhead required to provide primary storage within a pool to requesting threads.

Thread interruptions (data base and nondata base) is the total number of interruptions to threads (not necessarily assigned to this pool) which were required to transfer data into the pool to permit instruction execution. Note that on overflow, the machine resets the *thread interruptions (data base or non data base)* value from 2,147,483,647 back to 0 without any indication of error.

Data transferred to pool (data base and nondata base) is the amount of data transferred from auxiliary storage to the pool to permit instruction execution and as a consequence of set access state, implicit access group movement, and internal machine actions. Note that on overflow, the machine resets the *data transferred to pool (data base or non data base)* value from 2,147,483,647 back to 0 without any indication of error.

The **amount of pool not assigned to virtual addresses** represents the storage available to be used for new transfers into the main storage pool without displacing any virtual data already in the pool. The value returned will not include any storage that has been reserved for load/dump sessions active in the pool.

MPL Control Data with 2-byte counts (Hex 0A):

Note: Option hex 16 is the preferred method of materializing MPL control information. If option 0A is used and the actual value of any returned template field, other than transition counts, exceeds 32,767 then a value of 32,767 is returned (the values will not wrap). The transition counts are an exception and, as documented, do wrap after reaching their maximum value.

Offset Dec	Hex	Field Name	Data Type and Length	
16	10	Machine-wide MPL control	Char(16)	
16	10		Machine maximum number of MPL classes	Bin(2)
18	12		Machine current number of MPL classes	Bin(2)
20	14		MPL (max)	Bin(2)
22	16		Ineligible event threshold	Bin(2)
24	18		MPL (current)	Bin(2)
26	1A		Number of threads in ineligible state	Bin(2)
28	1C		Reserved	Char(4)
32	20	MPL class information	[*] Char(16)	
		(repeated for each MPL class, from 1 to the current number of MPL classes)		
32	20		MPL (max)	Bin(2)
34	22		Ineligible event threshold	Bin(2)
36	24		Current MPL	Bin(2)
38	26		Number of threads in ineligible state	Bin(2)
40	28		Number of threads assigned to class	Bin(2)
42	2A		Number of active to ineligible transitions	Bin(2)
44	2C		Number of active to MI wait transitions	Bin(2)
46	2E		Number of MI wait to ineligible transitions	Bin(2)
*	*	—- End —-		

Machine-Wide MPL Control:

Maximum number of MPL classes is the largest number of MPL classes allowed in the machine. These are assumed to be numbered from 1 to the maximum.

Current number of MPL classes is a user-specified value for the number of MPL classes in use. They are assumed to be numbered from 1 to the current number.

MPL (max) is the maximum number of processes which may concurrently be in the active state in the machine.

Ineligible event threshold is a number which, if exceeded by the *machine number of ineligible processes* defined below, will cause an event to be signaled.

MPL (current) is the current number of threads in the active state.

Number of threads in ineligible state is the number of threads not currently active because of enforcement of both the machine and class MPL rules.

MPL Class Information

MPL class information is data in an array that is associated with an MPL class by virtue of its ordinal position within the array.

MPL (max) is the number of threads assigned to the class which may be concurrently active.

Ineligible event threshold, **MPL (current)**, and **number of threads in ineligible state** are as defined above but apply only to threads assigned to the class.

Number of threads assigned to class is the total number of threads, in any state, assigned to the class.

The total number of transitions among the active, wait, and ineligible states by threads assigned to a class are:

1. **Number of active to ineligible transitions**
2. **Number of active to MI wait transitions**
3. **Number of MI wait to ineligible transitions**

Note that transitions from wait state to active state can be derived as (2 - 3) and transitions from ineligible state to active state as (1 + 3). On overflow, the machine wraps these Bin(2) numbers from 32,767 to 0 without any indication of error.

Machine Reserved Storage Pool Information (Hex 0C):

Offset Dec	Offset Hex	Field Name		Data Type and Length
16	10	Current number of pools		Bin(2)
18	12	Reserved		Char(6)
24	18	Individual main storage pool information (repeated once for each pool, up to the current number of pools)		[*] Char(16)
24	18		Pool size	Bin(4)
28	1C		Machine portion of the pool	Bin(4)
32	20		Reserved	Char(8)
*	*	—- End —-		

Pool size is the amount of main storage assigned to the pool (including the machine reserved portion).

Machine portion of the pool specifies the amount of storage from the pool that is dedicated to machine functions.

Both of the values above are in units equal to the *machine minimum transfer size*.

User storage area 1 - OBSOLETE (Hex 11):

This option is no longer used. The *number of bytes available for materialization* will always indicate that no user data is available.

Auxiliary Storage Information (Hex 12):

The *auxiliary storage information* describes the ASPs (auxiliary storage pools) which are configured within the machine and the units of auxiliary storage currently allocated to an ASP or known to the machine but not allocated to an ASP. This option does not return information for independent ASPs which are varied

off. You can use option "Auxiliary Storage information including offline Independent ASPs (Hex 20)" (page 883) to return information about independent ASPs which are varied off.

Also note that through appropriate setting of the number of bytes provided field for operand 1, the amount of information to be materialized for this option can be reduced thus avoiding the processing for unneeded information. As an example, by setting this field to only provide enough bytes for the common 16 byte header, plus the option hex 12 control information, plus the system ASP entry of the ASP information, you can get just the information up through the system ASP entry returned and avoid the overhead for the user ASPs and unit information.

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
16	10	Control information (occurs just once)	Char(64)	
16	10		Number of ASPs	Bin(2)
18	12		Number of allocated auxiliary storage units Note: Number of configured, non-mirrored units + number of mirrored pairs	Bin(2)
20	14		Number of unallocated auxiliary storage units	Bin(2)
22	16		Reserved (binary 0)	Char(2)
24	18		Maximum auxiliary storage allocated to temporaries	Char(8)
32	20		Reserved (binary 0)	Char(12)
44	2C		Unit information offset	Bin(4)
48	30		Number of pairs of mirrored units	Bin(2)
50	32		Mirroring main storage	Bin(4)
54	36		Number of multipath units	UBin(2)
56	38		Current auxiliary storage allocated to temporaries	Char(8)
64	40		Number of bytes in a page	Bin(4)
68	44		Number of independent ASPs	UBin(2)
70	46		Number of disk units in all varied on independent ASPs	UBin(2)
72	48		Number of basic ASPs	UBin(2)
74	4A		Number of disk units in all basic ASPs	UBin(2)
76	4C		Number of disk units in the system ASP	UBin(2)
78	4E		Number of additional entries for multipath units	UBin(2)
80	50	ASP information (Repeated once for each ASP. Located immediately after the control information above. ASP 1, always configured, is first. Configured user ASPs follow in ascending numerical order.)	[*] Char(160)	
80	50		ASP number	Char(2)
82	52		ASP control flags	Char(1)
82	52		Suppress threshold exceeded event	Bit 0
82	52		ASP overflow	Bit 1
82	52		Reserved	Bits 2-3
82	52		ASP mirrored	Bit 4
82	52		User ASP MI state	Bit 5

Offset Dec	Offset Hex	Field Name	Data Type and Length		
82	52		ASP overflow storage available		Bit 6
82	52		Suppress available storage lower limit reached event		Bit 7
83	53		ASP overflow recovery result	Char(1)	
83	53		Successful		Bit 0
83	53		Failed due to insufficient free space		Bit 1
83	53		Cancelled		Bit 2
83	53		Reserved (binary 0)		Bits 3-7
84	54		Number of allocated auxiliary storage units in ASP Note: Number of configured, non-mirrored units + number of mirrored pairs	UBin(2)	
86	56		Remote mirror performance mode **Hex 01 =** Synchronous mode **Hex 02 =** Asynchronous mode	Char(1)	
87	57		Remote mirror copy data state **Hex 00 =** Remote IASP mirroring is not configured **Hex 01 =** Remote copy is in sync with the production copy **Hex 02 =** Remote copy contains useable data **Hex 03 =** Remote copy data cannot be used	Char(1)	
88	58		ASP media capacity	Char(8)	
96	60		Reserved	Char(8)	
104	68		ASP space available	Char(8)	
112	70		ASP event threshold	Char(8)	
120	78		ASP event threshold percentage	Bin(2)	
122	7A		Additional ASP control flags	Char(2)	
122	7A		Terminate immediately when out of storage		Bit 0
122	7A		ASP contains compressed and non-compressed units		Bit 1
122	7A		Recover overflowed basic ASP during normal mode IPL		Bit 2
122	7A		Independent ASP		Bit 3
122	7A		ASP is online		Bit 4
122	7A		Independent ASP address threshold exceeded		Bit 5
122	7A		Independent ASP is remote mirrored		Bit 6
122	7A		Reserved (binary 0)		Bits 7-15
124	7C		ASP compression recovery policy	Char(1)	

Offset Dec	Hex	Field Name	Data Type and Length		
124	7C		Error recovery policy		Bits 0-1
			00 = Retry while space available		
			01 = Overflow immediately		
			10 = Retry forever		
124	7C		Reserved (binary 0)		Bits 2-7
125	7D		Independent ASP type	Char(1)	
125	7D		Primary ASP		Bit 0
125	7D		Secondary ASP		Bit 1
125	7D		UDFS ASP		Bit 2
125	7D		Reserved (binary 0)		Bits 3-7
126	7E		Remote mirror role	Char(1)	
			Hex 00 = Remote IASP mirroring is not configured		
			Hex 01 = System does not own a physical independent ASP copy		
			Hex 02 = Remote mirror role is unknown		
			Hex C4 = System owns a detached mirror copy		
			Hex D4 = System owns the mirror copy		
			Hex D7 = System owns the production copy		
127	7F		Remote mirror copy state	Char(1)	
			Hex 00 = Remote IASP mirroring is not configured		
			Hex 01 = System attempts to perform independent ASP remote mirroring when independent ASP is online.		
			Hex 02 = Remote independent ASP role is resuming.		
			Hex 03 = System is resuming and independent ASP is online and performing synchronization		
			Hex 04 = Remote independent ASP is detached and remote mirroring is not being performed.		
128	80		ASP system storage	Char(8)	
136	88		ASP overflow storage	Char(8)	
144	90		Space allocated to the error log	Bin(4)	
148	94		Space allocated to the machine log	Bin(4)	
152	98		Space allocated to the machine trace	Bin(4)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
156	9C		Space allocated for main store dump	Bin(4)
160	A0		Space allocated to the microcode	Bin(4)
164	A4		Remote mirror synchronization priority	Char(1)
			Hex 00 = Remote IASP mirroring is not configured	
			Hex 10 = Synchronization is given high priority	
			Hex 20 = Synchronization is given medium priority	
			Hex 30 = Synchronization is given low priority	
165	A5		Remote mirror encryption mode	Char(1)
			Hex 00 = Remote IASP mirroring is not configured	
			Hex 01 = Data being sent to remote mirror site is not encrypted	
			Hex 012= Data being sent to remote mirror site is encrypted	
166	A6		Remote mirror error recovery	Char(1)
			Hex 00 = Remote IASP mirroring is not configured	
			Hex 02 = Remote mirroring is suspended when an independent ASP error is detected.	
			Hex 03 = Remote mirroring is ended when an independent ASP error is detected.	
167	A7		Remote mirror minutes until timeout	Char(1)
168	A8		Available storage lower limit	Char(8)
176	B0		Protected space capacity	Char(8)
184	B8		Unprotected space capacity	Char(8)
192	C0		Protected space available	Char(8)
200	C8		Unprotected space available	Char(8)
208	D0		Reserved (binary 0)	Char(8)
216	D8		Number of addresses remaining in independent ASP	Char(8)
224	E0		Reserved (binary 0)	Char(16)
*	*	Unit information	[*] Char(208)	

Offset Dec	Offset Hex	Field Name	Data Type and Length		
		(Consists of one entry each for the configured, non-mirrored units and one unit of the mirrored pairs, the non-configured units, and the other unit of the mirrored pairs, and an entry for each multipath connection. An allocated storage unit (ASU) is either an allocated, non-mirrored unit or a mirrored pair. Note that the mirrored pair counts only as one ASU. When used without qualification, the term unit refers to an ASU. Unit information start may be located by the Unit Information Offset in the control information.)			
*	*		Device type	Char(8)	
*	*		Disk type		Char(4)
*	*		Disk model		Char(4)
*	*		Device identification	Char(8)	
*	*		Unit number		Char(2)
*	*		Reserved		Char(6)
*	*		Reserved	Char(4)	
*	*		Unit ASP number	Char(2)	
*	*		Logical mirrored pair status	Char(1)	
*	*		Unit mirrored		Bit 0
*	*		Mirrored unit protected		Bit 1
*	*		Mirrored pair reported		Bit 2
*	*		Reserved		Bits 3-7
*	*		Mirrored unit status	Char(1)	
*	*		Unit media capacity	Char(8)	
*	*		Unit storage capacity	Char(8)	
*	*		Unit space available	Char(8)	
*	*		Unit space reserved for system	Char(8)	
*	*		Reserved	Char(6)	
*	*		Unit control flags	Char(2)	
*	*		Reserved (binary 0)		Bit 0
*	*		Unit is device parity protected		Bit 1
*	*		Subsystem is active		Bit 2
*	*		Unit in subsystem has failed		Bit 3
*	*		Other unit in subsystem has failed		Bit 4
*	*		Subsystem runs in degraded mode		Bit 5
*	*		Hardware failure		Bit 6
*	*		Device parity protection is being rebuilt		Bit 7
*	*		Unit is not ready		Bit 8
*	*		Unit is write protected		Bit 9
*	*		Unit is busy		Bit 10
*	*		Unit is not operational		Bit 11
*	*		Status is not recognizable		Bit 12
*	*		Status is not available		Bit 13
*	*		Unit is read/write protected		Bit 14
*	*		Unit is compressed		Bit 15
			Bits 2 to 14 are mutually exclusive.		

Offset Dec	Offset Hex	Field Name	Data Type and Length		
*	*		Additional unit control flags	Char(2)	
*	*			Do not allocate additional storage on this disk unit	Bit 0
*	*			Unit is in availability parity set	Bit 1
*	*			Unit is multipath unit	Bit 2
*	*			Reserved (binary 0)	Bits 3-15
*	*		Reserved (binary 0)	Char(14)	
*	*		Reserved (binary 0)	Char(42)	
*	*		Unit Identification	Char(22)	
*	*			Serial Number	Char(1
*	*			Resource name	Char(1
*	*			Reserved (binary 0)	Char(2
*	*		Unit usage information	Char(64)	
*	*			Blocks transferred to main storage	Bin(4)
*	*			Blocks transferred from main storage	Bin(4)
*	*			Requests for data transfer to main storage	Bin(4)
*	*			Requests for data transfer from main storage	Bin(4)
*	*			Permanent blocks transferred from main storage	Bin(4)
*	*			Requests for permanent data transfer from main storage	Bin(4)
*	*			Reserved (binary 0)	Char(8
*	*			Sample count	Bin(4)
*	*			Not busy count	Bin(4)
*	*			Reserved (binary 0)	Char(2
*	*	—- End —-			

Number of ASPs is the number of ASPs configured within the machine. One, the minimum value, indicates just the system ASP exists and that there are no user ASPs configured. Up to 254 user ASPs can be configured. The system ASP always exists. The *number of ASPs* includes the system ASP, user ASPs which are basic ASPs (that is, user ASPs which cannot be varied on or off), and independent ASPs which are currently varied on to this system.

Number of allocated auxiliary storage units is the total number of configured units logically addressable by the system as units. This is the number of configured, non-mirrored units plus the number of mirrored pairs allocated to the ASPs. This number includes only the first path of a multipath connection unit. The count of the remaining paths connected to multipath units is materialized in *number of additional entries for multipath units*. The total number of disk actuator arms on the system is the sum of the allocated auxiliary storage units plus the number of unallocated auxiliary storage units plus the number of pairs of mirrored units. Information on these units is materialized as part of the unit information. Any two units of the same size may be associated to form a mirrored pair. Association of two units as a mirrored pair reduces the amount of logically available storage by the number of bytes contained on one of the mirrored units in the mirrored pair. The disk units reside in the system ASP, a basic ASP, or an independent ASP. This number specifies the number of *unit information* entries that can be materialized.

Number of unallocated auxiliary storage units is the number of auxiliary storage units that are currently not allocated to an ASP. Information on these units is materialized as part of the unit information.

Maximum auxiliary storage allocated to temporaries is the maximum number of bytes of temporary storage allocated at any one time since the last IPL of the machine. This includes the temporary storage allocated on the load source unit.

Unit information offset is the offset, in bytes, from the start of the operand 1 materialization template to the start of the unit information. This value can be added to a space pointer addressing the start of operand 1 to address the start of the unit information.

Number of pairs of mirrored units represents the number of mirrored pairs in the system. Each mirrored pair consists of two mirrored units; however, only one of the two mirrored units is guaranteed to be operational.

Mirroring main storage is the number of bytes of main storage in the machine storage pool used by mirroring. This increases when mirror synchronization is active. This amount of storage is directly related to the number of mirrored pairs.

Number of multipath units is the number of disk units that have multiple connections to a disk unit. This means that there are multiple resource names that all represent the same disk unit, yet each represents a unique path to the disk unit. All active connections will be used for communicating with the disk unit.

Current auxiliary storage allocated to temporaries is the number of bytes of temporary storage allocated on the system. This includes the temporary storage allocated on the load source unit.

Number of bytes in a page is the number of bytes in a single page. This can be used to convert fields that are given in pages into the correct number of bytes.

Number of independent ASPs is the number of independent ASPs varied on to this system. An independent ASP is an ASP that can be varied on or off.

Number of disk units in all varied on independent ASPs is the number of configured units logically addressable by all independent ASPs which are currently varied on to this system. Information on these units is materialized as part of the unit information.

Number of basic ASPs is the number of basic ASPs configured on this system. A basic ASP is a user ASP that cannot be varied on or off.

Number of disk units in all basic ASPs is the total number of configured units logically addressable by all basic ASPs. Information on these units is materialized as part of the unit information.

Number of disk units in the system ASP is the total number of configured units logically addressable in the system ASP. Information on these units is materialized as part of the unit information.

Number of additional entries for multipath units is the number of additional unit entries that can be materialized for the multipath connection devices. The first path of each unit is not included in this total.

ASP information is repeated once for each configured ASP within the machine that is online. The number of ASPs configured is specified by the *number of ASPs* field. ASP 1, the system ASP, is materialized first. Because the system ASP always exists, its materialization is always available. The user ASPs which are configured are materialized after the system ASP in ascending numerical order. There may be gaps in the numerical order. That is, if just user ASPs 3 and 5 are configured, only information for them is materialized producing information on just ASP 1, ASP 3 and ASP 5 in that order.

ASP number uniquely identifies the auxiliary storage pool. The ASP number may have a value from 1 through 255. A value of 1 indicates the system ASP. A value of 2 through 255 indicates a user ASP. Note that independent ASPs have a value of 33 through 255.

Suppress threshold exceeded event flag indicates whether or not the machine is suppressing signaling of the related event when the event threshold in effect for this ASP has been exceeded. A value of binary 1 indicates that the signaling is being suppressed; binary 0 indicates that the signaling is not being suppressed. The default after each IPL of the machine is that the signaling is not suppressed; i.e. default is binary 0. For the system ASP, this flag is implicitly set to binary 1 by the machine when the machine auxiliary storage threshold is exceeded. For a basic ASP, this flag is implicitly set to binary 1 by the machine when the user auxiliary storage threshold is exceeded.

The **ASP overflow** flag indicates whether or not object allocations directed into a basic ASP have overflowed into the system ASP. A value of binary 1 indicates overflow; binary 0 indicates no overflow. This flag does not apply to the system ASP and a value of binary 0 is always returned for it. This flag does not apply to independent ASPs and a value of binary 0 is always returned for independent ASPs.

ASP mirrored flag specifies whether or not the ASP is configured to be mirror protected. A value of binary 1 indicates that ASP mirror protection is configured. Refer to the *mirrored unit protected* flag to determine if mirror protection is active for each mirrored pair. A value of binary 0 indicates that none of the units associated with the ASP are mirrored.

User ASP MI state indicates the state of the user ASP. A value of binary 1 indicates that the user ASP is in the 'new' state. This means that a context may be allocated in this user ASP. A value of binary 0 indicates that the user ASP is in the 'old' state. This means that there are no contexts allocated in this user ASP. This flag has no meaning for the system ASP and a value of binary 0 will always be returned. A value of binary 1 will always be returned for independent ASPs.

ASP overflow storage available flag indicates whether or not the amount of auxiliary storage that has overflowed from the basic ASP into the system ASP is available. A value of binary 1 indicates that the amount is maintained by the machine and available in the *ASP overflow storage* field. A value of binary 0 indicates that the amount is not available. This flag does not apply to independent ASPs and a value of binary 0 is always returned for independent ASPs.

Suppress available storage lower limit reached event flag indicates whether the machine will signal the related event when the available storage lower limit has been reached. This field currently has meaning only in the system ASP (ASP 1). This value will always be returned as binary 0 for a user ASP. A value of binary 1 indicates that signaling of the event is being suppressed; binary 0 indicates that signaling of the event is not suppressed. The default after each IPL of the machine is binary 0, i.e., signaling of this event is not suppressed. This flag is set to binary 1 by the machine when the *available storage lower limit reached* (hex 000C,02,08) event is signaled. This is done to avoid repetitive signaling of the event when the available storage lower limit reached condition occurs.

ASP overflow recovery result flags indicate the result of the ASP overflow recovery operation which is performed during an IPL upon request by the user. When this operation is requested, the machine attempts to recover a basic ASP from an overflow condition by moving overflowed auxiliary storage from the system ASP back to the basic ASP during the Storage Management recovery step of an IPL. The *successful* flag is set to a value of binary 1 when all the overflowed storage was successfully moved. The *failed due to insufficient free space* flag is set to a value of binary 1 when there is not sufficient free space in the basic ASP to move all the overflowed storage. The *cancelled* flag is set to a value of binary 1 when the operation was cancelled prior to completion (e.g., system power loss, user initiated IPL). This flag does not apply to independent ASPs and a value of binary 0 is always returned for independent ASPs.

Number of allocated auxiliary storage units in ASP is the number of configured units logically addressable by the system as units for this ASP. This is the number of configured, non-mirrored units plus the number of mirrored pairs allocated in the ASPs. The total number of units (actuator arms) on the system is the sum of the allocated auxiliary storage units plus the number of unallocated auxiliary storage units plus the number of pairs of mirrored units. For example, each 9335 enclosure represents two units. Information on these units is materialized as part of the unit information. Any two units of the

same size may be associated to form a mirrored pair. Association of two units as a mirrored pair reduces the amount of logically available storage by the number of bytes contained on one of the mirrored units in the mirrored pair.

Remote mirror performance mode specifies the mode in which remote mirroring operates. A value of hex 01 indicates synchronous mode. In synchronous mode, the client waits for the operation to complete on both the source and on the target. A value of hex 02 indicates aynchronous mode. In asynchronous mode, the client waits for the operation to complete on the source and for the operation to be received on the target.

Remote mirror copy data state specifies the condition of the data on the target. A value of hex 00 indicates that remote independent ASP mirroring is not configured. A value of hex 01 indicates that the remote copy is absolutely in sync with the production copy. A value of hex 02 indicates that the remote copy contains usable data. A detached mirror copy always has usable data state. A value of hex 03 indicates that there is incoherent data state in the mirror copy and the data cannot be used.

ASP media capacity specifies the total space, in number of bytes of auxiliary storage, on the storage media allocated to the ASP. This is just the sum of the unit media capacity fields for (1) the units allocated to the ASP or (2) the mirrored pairs in the ASP.

ASP space available is the number of bytes of auxiliary storage that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation in the ASP. Note that a mirrored pair counts for only one unit.

ASP event threshold specifies the minimum value for the number of bytes of auxiliary storage available in the ASP prior to the exceeded condition occurs when the ASP space available value becomes equal to or less than the ASP event threshold value. Refer to the definition of the *suppress threshold exceeded event* flag for more information.

The *ASP event threshold* value is calculated from the ASP event threshold percentage value by multiplying the ASP media capacity value by the ASP event threshold percentage and subtracting the product from that same capacity value.

ASP event threshold percentage specifies the auxiliary storage space utilization threshold as a percentage of the ASP media capacity. This value is used, as described above, to calculate the ASP event threshold value. This value can be modified through use of Dedicated Service Tool DASD configuration options.

Terminate immediately when out of storage indicates whether the system will be terminated immediately when a request for space occurs in the system ASP that cannot be satisfied because the system is out of storage. A value of binary 1 indicates that when a request for space in the system ASP cannot be satisfied, then the system will be terminated immediately. This field currently has meaning only in the system ASP (ASP 1). This value will always be returned as binary 0 for a user ASP.

Note:

For a physical machine with firmware level hex 00, when a request for space in the system ASP cannot be satisfied in the primary partition and the value for *terminate immediately when out of storage* is binary 1 in the primary partition, all partitions in the physical machine will terminate. When a request for space in the system ASP cannot be satisfied in a secondary partition and the value for *terminate immediately when out of storage* is binary 1 in that partition, *only* the partition in which the condition occurred will terminate. MATMATR option hex 01E0 can be used to materialize the firmware level.

For a physical machine with firmware level hex 10, *only* the partition in which the condition occurred will terminate.

A value of binary 0 indicates that when a request for space in the system ASP cannot be satisfied, then the system will not be terminated immediately, but will be allowed to continue to run however it can.

ASP contains compressed and non-compressed units flag specifies whether or not the ASP has compressed and non-compressed configured units. A value of binary 1 indicates that both compressed and non-compressed units exist in this ASP. A value of binary 0 indicates that a mix of compressed and non-compressed units does not exist in this ASP.

Recover overflowed basic ASP during normal mode IPL flag specifies whether or not the machine will attempt to recover the overflowed ASP data during normal mode IPLs. Overflowed data is data from the basic ASP which exists in the system ASP because there was insufficient auxiliary storage in the basic ASP. A value of binary 1 indicates that the machine will attempt to automatically recover any overflowed data for that basic ASP during normal mode IPLs. A value of binary 0 indicates that the machine will not attempt to recover the overflowed data. A value of binary 0 is always returned for the system ASP (ASP 1). A value of binary 0 is always returned for an independent ASP (since an independent ASP can never overflow its data into the system ASP).

Independent ASP flag specifies whether or not the ASP is an independent ASP; that is, it can be varied on and off. A value of binary 1 indicates the ASP is an independent ASP. A value of binary 0 indicates that this ASP is a basic ASP or the system ASP and cannot be varied on or off.

ASP is online flag always returns the value of binary 1.

Independent ASP address threshold exceeded flag is only valid for an Independent ASP and specifies whether or not the independent ASP address threshold, selected by the machine, has been exceeded. A value of binary 1 indicates the threshold has been exceeded and the Independent ASP is running low on addresses. A value of binary 0 indicates that the address threshold has not been exceeded.

Independent ASP is remote mirrored indicates that the independent ASP is remote mirrored. Remote independent ASP mirroring provides high availability by supporting multiple physical independent ASP copies at different sites that contain the same user data with the same virtual addresses. A value of binary 0 indicates that the independent ASP is not remote mirrored. A value of binary 1 indicates that the independent ASP is remote mirrored.

ASP compression recovery policy indicates how Storage Management handles a failure condition due to a compressed disk unit being temporarily full as auxiliary storage space is reserved on the unit.

A value of binary 00 indicates that if the I/O processor can make storage space available by rearranging and further compressing data on the unit, Storage Management waits for space to be made available. When the I/O processor makes sufficient space on the compressed unit to contain the Storage Management request, the request completes successfully and the system resumes normal processing. If space can not be made available on the unit, auxiliary storage overflows from the basic ASP to the system ASP.

A value of binary 01 indicates that auxiliary storage overflows from the user ASP to the system ASP. Storage Management does not wait for the I/O processor to make storage space available on the unit.

A value of binary 10 indicates that Storage Management waits indefinitely for storage space to be made available on the unit, even if the I/O processor can not make space available on the unit. No auxiliary storage overflows from the user ASP to the system ASP.

A value of binary 00 is always returned for the system ASP (ASP 1). A value of binary 10 is always returned for independent ASPs (that is, for ASPs which can be varied on or off). An independent ASP can never have a value of binary 01 (overflow immediately) because independent ASPs are not allowed to overflow into the system ASP.

Primary ASP flag indicates that the independent ASP is a primary ASP in an ASP group. A primary ASP defines a collection of directories and contexts and may have secondary ASPs associated with it. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a primary ASP. A value of binary 0 indicates the independent ASP is not a primary ASP.

Secondary ASP flag indicates that the independent ASP is a secondary ASP in an ASP group. A secondary ASP is associated with a primary ASP. There can be many secondary ASPs associated with the same primary ASP. The secondary ASP defines a collection of directories and contexts. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a secondary ASP. A value of binary 0 indicates the independent ASP is not a secondary ASP.

UDFS ASP flag indicates that the independent ASP is a UDFS (User-defined File System) ASP. This type of independent ASP cannot be a member of an ASP group. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a UDFS ASP. A value of binary 0 indicates the independent ASP is not a UDFS ASP.

Remote mirror role identifies the current role of the physical independent ASP copy. A value of hex 00 indicates that remote independent ASP mirroring is not configured. A value of hex 01 indicates that the system does not own a physical independent ASP copy. A value of hex 02 indicates that the remote mirror role is unknown. A value of hex D7 indicates that the system owns the production copy. A value of hex D4 indicates that the system owns the mirror copy. A value of hex C4 indicates that the system owns a detached mirror copy.

Remote mirror copy state identifies the mirror state of the mirror copy. A value of hex 00 indicates that remote independent ASP mirroring is not configured. A value of hex 01 indicates that the system attempts to perform independent ASP remote mirroring when it is online. A value of hex 02 indicates that the remote independent ASP role is resuming, but the independent ASP is offline so it is not performing synchronization. A value of hex 03 indicates that the system is resuming and the independent ASP is online, so it is performing synchronization. A value of hex 04 indicates that the remote independent ASP role is detached and remote mirroring is not being performed.

ASP system storage specifies the amount of system storage currently allocated in the ASP in bytes.

ASP overflow storage indicates the number of bytes of auxiliary storage that have overflowed from a basic ASP into the system ASP. This value is valid only if the *ASP overflow storage available* field is set to a value of binary 1.

Space allocated to the error log is the number of pages of auxiliary storage that are allocated to the error log. This field only applies to the system ASP.

Space allocated to the machine log is the number of pages of auxiliary storage that are allocated to the machine log. This field only applies to the system ASP.

Space allocated to the machine trace is the number of pages of auxiliary storage that are allocated to the machine trace. This field only applies to the system ASP.

Space allocated for main store dump is the number of pages of auxiliary storage that are allocated to the main store dump space. The contents of main store are written to this location for some system terminations. This field only applies to the system ASP.

Space allocated to the microcode is the number of pages of auxiliary storage that are allocated for microcode and space used by the microcode. The space allocated to the error log, machine log, machine trace, and main store dump space is not included in this field.

Remote mirror synchronization priority indicates the priority assigned to synchronization between the physical copy and the mirrored copy related to work on the system. A value of hex 00 indicates that

remote independent ASP mirroring is not configured on this independent ASP. A value of hex 10 indicates that the synchronization is given high priority, and is completed quickly at the expense of significant degradation to work on the system. A value of hex 20 indicates that the synchronization is given medium priority, and is completed at a moderate rate with some degradation to work on the system. A value of hex 30 indicates that the synchronization is given low priority, and is completed at a slow rate with minimum degradation to work on the system.

Remote mirror encryption mode indicates the encryption mode for the remote mirrored independent ASP. A value of hex 00 indicates that remote independent ASP mirroring is not configured on this independent ASP. A value of hex 01 indicates that the user has chosen not to encrypt the data being sent to the remote mirror site. A value of hex 02 indicates that the user has chosen to encrypt the data being sent to the remote mirror site.

Remote mirror error recovery policy indicates the error recovery policy selected by the user. A value of hex 00 indicates that remote independent ASP mirroring is not configured on this system. A value of hex 02 indicates that remote mirroring is suspended when an IASP error is detected. After suspend, if the target node becomes accessible, the system automatically resumes remote independent ASP mirroring. A value of hex 03 indicates that remote mirroring is ended when an IASP error is detected.

Remote mirror minutes until timeout is the number of minutes the system waits for a write acknowledgement from the remote system before the error recovery policy selected by the user is implemented.

Available storage lower limit is the number of bytes of available auxiliary storage in the system ASP prior to the *available storage lower limit reached* condition occurring. When the amount of auxiliary storage available in the system ASP becomes less than this amount, the *available storage lower limit reached* (hex 000C,02,08) event is signaled if it is not suppressed. Redundant signaling of this event is suppressed as indicated by the setting of the *suppress available storage lower limit reached* event flag.

Protected space capacity specifies the total number of bytes of auxiliary storage that is protected by mirroring or device parity in the ASP. Note that a varied-off independent ASP could have 0 in this field because the system could not determine what disk units exist in a varied-off independent ASP .

Unprotected space capacity specifies the total number of bytes of auxiliary storage that is not protected by mirroring or device parity in the ASP. Note that a varied-off independent ASP could have 0 in this field because the system could not determine what disk units exist in a varied-off independent ASP .

Protected space available specifies the number of bytes of protected auxiliary storage that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation in the ASP. Note that a varied-off independent ASP could have 0 in this field because the system could not determine what disk units exist in a varied-off independent ASP .

Unprotected space available specifies the number of bytes of unprotected auxiliary storage that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation in the ASP. Note that a varied-off independent ASP could have 0 in this field because the system could not determine what disk units exist in a varied-off independent ASP .

Number of addresses remaining in independent ASP contains the number of virtual addresses remaining for use by the independent ASP. This field only has meaning for an independent ASP. The information in this field is only valid if the *independent ASP address threshold exceeded* flag is set to binary 1.

Unit information is materialized in the following order:

Group 1: Configured units consisting of non-mirrored units and the first subunit of a pair of mirrored units.

Group 2: Non-configured units.

Group 3: Configured units consisting of the mates of mirrored units listed in group 1 (above).

The **unit information** is located by the **unit information offset** field which specifies the offset from the beginning of the operand 1 template to the start of the unit information. The number of entries for each of the three groups listed above is defined as follows:

Group 1: Number of non-mirrored, configured units + number of mirrored pairs

Group 2: Number of non-configured storage units (also called unallocated units).

Group 3: Number of mirrored pairs

For unallocated units the following fields contain meaningful information: device type, device identification, unit identification, unit control flags, unit relationship, and unit media capacity. The remaining fields have no meaning for unallocated units because the units are not currently in use by the system. Mirrored unit entries contain either current or last known information. The last known data consists of the mirrored unit status, disk type, disk model, unit ASP number, disk serial number, and unit address. Last known information is provided when the *mirrored pair reported* field is a binary 0.

Disk type identifies the type of disk enclosure containing this auxiliary storage unit. This is the four byte character field from the vital product data for the disk device which identifies the type of drive. For example, the value is character string '6607' for a 6607 device.

Disk model identifies the model of the type of disk enclosure containing this auxiliary storage unit. This is the four byte character field from the vital product data for the disk device which identifies the model of the drive.

Unit number uniquely identifies each non-mirrored unit or mirrored pair among the configured units. Both mirrored units of a mirrored pair have the same unit number. The value of the unit number is assigned by the system when the unit is allocated to an ASP. For unallocated units, the unit number is set to binary 0.

Unit ASP number specifies the ASP to which this unit is currently allocated. A value of 0 indicates that this unit is currently unallocated. A value of 1 specifies the system ASP. A value from 2 through 255 specifies a user ASP and correlates to the *ASP number* field in the *ASP information* entries. Values 33 to 255 specify a independent ASP. Values 2 to 32 specify a basic ASP.

Unit mirrored flag indicates that this unit number represents a mirrored pair. This bit is materialized with both mirrored units of a mirrored pair.

Mirrored unit protected flag indicates the mirror status of a mirrored pair. A value of 1 indicates that both mirrored units of a mirrored pair are active. A 0 indicates that one mirrored unit of a mirrored pair is not active. Active means that both units are on line and fully synchronized (i.e. the data is identical on both mirrored units).

Mirrored pair reported flag indicates that a mirrored unit reported as present. The mirrored unit reported present during or following IMPL. Current attachment of a mirrored unit to the system **cannot** be inferred from this bit. A 0 indicates that the mirrored unit being materialized is missing. The last known information pertaining to the missing mirrored unit is materialized. A 1 indicates that the mirrored unit being materialized has reported. The information for this reported unit is current to the last time it reported status to the system.

Mirrored unit status indicates mirrored unit status.

A value of 1 indicates that this mirrored unit of a mirrored pair is active (i.e. on-line with current data).

A value of 2 indicates that this mirrored unit is being synchronized.

A value of 3 indicates that this mirrored unit is suspended.

Mirrored unit status is stored as binary data and is valid only when the *unit mirrored* flag is on.

Unit media capacity is the space, in number of bytes of auxiliary storage, on the non-mirrored unit or mirrored pair, that is, the capacity of the unit prior to any formatting or allocation of space by the system it is attached to. For a mirrored pair, this space is the number of bytes of auxiliary storage on either one of the mirrored units. The space is identical on both of the mirrored units. Caution, do not attempt to add the capacities of the two units of a mirrored pair together. Unit media capacity is also known as ″logical capacity″. For compressed drives, the logical capacity is dynamic, and changes, depending on how well the data is compressed. A typical compressed logical capacity might be twice the drive's physical capacity.

Unit storage capacity has the same value as the *unit media capacity* for configured disk units. This value is 0 for non-configured units.

Unit space available is the number of bytes of secondary storage space that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation on the unit (or the mirrored pair). For a mirrored pair, this space is the number of bytes of auxiliary storage available on either one of the mirrored units. The space is identical on both of the mirrored units. Caution, do not attempt to add the capacities of the two units of a mirrored pair together. This value is 0 for non-configured units.

Unit space reserved for system is the total number of bytes of auxiliary storage on the unit reserved for use by the machine. This storage is not available for storing objects, redundancy data, and other internal machine data. This value is 0 for non-configured units.

Unit is device parity protected - a value of 1 indicates that this unit is device parity protected.

Subsystem is active indicates whether the array subsystem is active.

If the **unit in subsystem has failed** field is binary 1, the unit in an array subsystem being addressed has failed. Data protection for this subsystem is no longer in effect.

If the **other unit in subsystem has failed** field is binary 1, the unit being addressed is operational, but another unit in the array subsystem has failed. Data protection for this subsystem is no longer in effect.

If the **subsystem runs in degraded mode** field is binary 1, the array subsystem is operational and data protection for this subsystem is in effect, but a failure that may affect performance has occurred. It must be fixed.

If the **hardware failure** field is binary 1, the array subsystem is operational and data protection for this subsystem is in effect, but hardware failure has occurred. It must be fixed.

If the **device parity protection is being rebuilt** field is 1, the device parity protection for this device is being rebuilt following a repair action.

If the **unit is not ready** field is binary 1, the unit being addressed is not ready for I/O operation.

If the **unit is write protected** field is binary 1, the write operation is not allowed on the unit being addressed.

If the **unit is busy** field is binary 1, the unit being addressed is busy.

If the **unit is not operational** field is binary 1, the unit being addressed is not operational. The status of the device is not known.

If the **unit is not recognizable** field is binary 1, the unit being addressed has an unexpected status. I.e. the unit is operational, but its status returned to Storage Management from the IOP is not one of those previously described.

If the **status is not available** field is binary 1, the machine is not able to communicate with I/O processor. The status of the device is not known.

If the **unit is read/write protected** is binary 1, a DASD array may be in the read/write protected state when there is a problem, such as a cache problem, configuration problem, or some other array problems that could create a data integrity exposure.

If the **unit is compressed** field is binary 1, the logical capacity of the unit may be greater than its physical capacity in bytes, depending on how well the data can be compressed.

If the **do not allocate additional storage on this disk unit** field is binary 1, then new allocations will be directed away from this unit.

If the **unit is in availability parity set** field is binary 1, the unit being addressed is in a parity set optimized for availability.

If the **unit is multipath unit** field is binary 1, the unit being addressed has multipath connections to the disk unit.

Serial number specifies the serial number of the device containing this auxiliary storage unit. This is the ten character serial number field from the vital product data for the disk device.

Resource name is the unique ten-character name assigned by the system

Unit usage information specifies statistics relating to usage of the unit. For unallocated units, these fields are meaningless.

Blocks transferred to/from main storage fields specify the number of 512-byte blocks transferred for the unit since the last IMPL. These values wrap around to zero and continue counting in the case of an overflow of the field with no indication of the overflow having occurred.

Requests for data transfer to/from main storage fields specify the number of data transfer (I/O) requests processed for the unit since the last IMPL. These values wrap around to zero and continue counting in the case of an overflow of the field with no indication of the overflow having occurred. These values are not directly related to the number of blocks transferred for the unit because the number of blocks to be transferred for a given transfer request can vary greatly.

Permanent blocks transferred from main storage specifies the number of 512-byte blocks of permanent data transferred from main storage to auxiliary storage for the unit since the last IMPL. In the case of an overflow of the field, this value wraps around back to zero and continues counting, with no indication of the overflow condition having occurred.

Requests for permanent data transfer from main storage specifies the number of transfer (I/O) requests for transfers of permanent data from main storage to auxiliary storage that have been processed for the unit since the last IMPL. In the case of an overflow of the field, this value wraps around back to zero and continues counting, with no indication of the overflow condition having occurred. This value is not directly related to the permanent blocks transferred from main storage value for the unit ASP because the number of blocks to be transferred for any particular transfer request can vary greatly.

Sample count specifies the number of times the disk queue was checked to determine whether or not the queue is empty.

Not busy count specifies the number of times the disk queue was empty during the same time period that the sample count was taken.

Note that on overflow, the machine resets the following BIN(4) fields from 2,147,483,647 back to 0 without any indication of error: *blocks transferred to main storage, blocks transferred from main storage, requests for data transfer to main storage, requests for data transfer from main storage, permanent blocks transferred from main storage, requests for permanent data transfer from main storage, sample count,* and *not busy count*.

Original Multiprocessor utilizations (Hex 13):

Note: Option hex 28 is the preferred method of materializing multiprocessor utilizations. The MATRMD instruction option hex 28 returns the same information (and more, in a different format) as that provided by this option. This option returns information on a maximum of 32 processors even if there are more processors installed. If information on more than 32 processors is required, then the hex 28 option must be used.

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
16	10	Maximum number of active processors in the partition	Bin(2)	
18	12	Number of active processors in the partition	Bin(2)	
20	14	Bit map of processors currently active on the machine	Char(4)	
20	14		Processor 1 is active	Bit 0
20	14		Processor 2 is active	Bit 1
20	14		Processor 3 is active	Bit 2
20	14		Processor 4 is active	Bit 3
20	14		Processor 5 is active	Bit 4
20	14		Processor 6 is active	Bit 5
20	14		Processor 7 is active	Bit 6
20	14		Processor 8 is active	Bit 7
20	14		Processor 9 is active	Bit 8
20	14		Processor 10 is active	Bit 9
20	14		Processor 11 is active	Bit 10
20	14		Processor 12 is active	Bit 11
20	14		Processor 13 is active	Bit 12
20	14		Processor 14 is active	Bit 13
20	14		Processor 15 is active	Bit 14
20	14		Processor 16 is active	Bit 15
20	14		Processor 17 is active	Bit 16
20	14		Processor 18 is active	Bit 17

Offset Dec	Hex	Field Name	Data Type and Length	
20	14		Processor 19 is active	Bit 18
20	14		Processor 20 is active	Bit 19
20	14		Processor 21 is active	Bit 20
20	14		Processor 22 is active	Bit 21
20	14		Processor 23 is active	Bit 22
20	14		Processor 24 is active	Bit 23
20	14		Processor 25 is active	Bit 24
20	14		Processor 26 is active	Bit 25
20	14		Processor 27 is active	Bit 26
20	14		Processor 28 is active	Bit 27
20	14		Processor 29 is active	Bit 28
20	14		Processor 30 is active	Bit 29
20	14		Processor 31 is active	Bit 30
20	14		Processor 32 is active	Bit 31
24	18	Array of Char(8) processor time used since IPL values. Repeated once for each active processor.	Char(256)	
24	18		Processor 1 time busy since IPL	Char(8)
32	20		Processor 2 time busy since IPL	Char(8)
40	28		Processor 3 time busy since IPL	Char(8)
48	30		Processor 4 time busy since IPL	Char(8)
56	38		Processor 5 time busy since IPL	Char(8)
64	40		Processor 6 time busy since IPL	Char(8)
72	48		Processor 7 time busy since IPL	Char(8)
80	50		Processor 8 time busy since IPL	Char(8)
88	58		Processor 9 time busy since IPL	Char(8)
96	60		Processor 10 time busy since IPL	Char(8)
104	68		Processor 11 time busy since IPL	Char(8)
112	70		Processor 12 time busy since IPL	Char(8)
120	78		Processor 13 time busy since IPL	Char(8)
128	80		Processor 14 time busy since IPL	Char(8)
136	88		Processor 15 time busy since IPL	Char(8)
144	90		Processor 16 time busy since IPL	Char(8)
152	98		Processor 17 time busy since IPL	Char(8)
160	A0		Processor 18 time busy since IPL	Char(8)
168	A8		Processor 19 time busy since IPL	Char(8)
176	B0		Processor 20 time busy since IPL	Char(8)
184	B8		Processor 21 time busy since IPL	Char(8)
192	C0		Processor 22 time busy since IPL	Char(8)
200	C8		Processor 23 time busy since IPL	Char(8)
208	D0		Processor 24 time busy since IPL	Char(8)
216	D8		Processor 25 time busy since IPL	Char(8)

Offset Dec	Hex	Field Name	Data Type and Length	
224	E0		Processor 26 time busy since IPL	Char(8)
232	E8		Processor 27 time busy since IPL	Char(8)
240	F0		Processor 28 time busy since IPL	Char(8)
248	F8		Processor 29 time busy since IPL	Char(8)
256	100		Processor 30 time busy since IPL	Char(8)
264	108		Processor 31 time busy since IPL	Char(8)
272	110		Processor 32 time busy since IPL	Char(8)
280	118	—- End —-		

This option always returns a *number of bytes available for materialization* equal to the length of the entire structure detailed above (it does not vary with the number of configured or active processors).

Maximum number of active processors in the partition is the maximum number of virtual processors that can be active on the current IPL of the partition.

Number of active processors in the partition is the number of virtual processors currently active in the partition. It will always be less than or equal to the *maximum number of active processors in the partition*.

A value of binary 1 for the **processor is active** field indicates the processor is active. A value of binary 0 indicates the processor is currently varied off or is not installed on the system.

The significance of bits within the **processor time busy since IPL** fields are the same as that defined for the time-of-day clock. For a partition sharing physical processors, *processor time busy since IPL* is scaled appropriately so that CPU utilization calculations can be done as if the partition was using dedicated processors.

Virtual processors that are not currently active (but were active at some previous time in the IPL) will not have their *processor time busy since IPL* reported.

Storage pool tuning (Hex 14):

Offset Dec	Hex	Field Name	Data Type and Length	
16	10	Control information (occurs just once)	Char(16)	
16	10		Current number of pools	Bin(2)
18	12		Reserved (binary 0)	Char(14)
32	20	Pool information (repeated once for each pool)	[*] Char(104)	
32	20		Type of pool tuning **Hex 00 =** No tuning is being done for the pool **Hex 10 =** Static tuning **Hex 20 =** Dynamic tuning of transfers to main storage **Hex 30 =** Dynamic tuning of transfers to main storage and to auxiliary storage	Char(1)
33	21		Changed page handling	Char(1)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			Hex 00 = System page replacement algorithm handles changed pages	
			Hex 10 = Periodically transfer changed pages to auxiliary storage	
34	22		Reserved (binary 0)	Char(14)
48	30		Nondatabase objects	Char(8)
48	30		Blocking factor	Char(2)
			Hex 0008 = Transfer data between main storage and auxiliary in blocks of 4K.	
			Hex 0010 = Transfer data between main storage and auxiliary in blocks of 8K.	
			Hex 0020 = Transfer data between main storage and auxiliary in blocks of 16K.	
			Hex 0040 = Transfer data between main storage and auxiliary in blocks of 32K.	
50	32		Reserved (binary 0)	Char(6)
56	38		Reserved (binary 0)	Char(16)
72	48		Handling of database objects by class	[4] Char(8)
			(repeat for each of the four classes)	
72	48		Blocking factor	Char(2)

Offset Dec	Offset Hex	Field Name	Data Type and Length
			Hex 0008 = Transfer data between main storage and auxiliary in blocks of 4K.
			Hex 0010 = Transfer data between main storage and auxiliary in blocks of 8K.
			Hex 0020 = Transfer data between main storage and auxiliary in blocks of 16K.
			Hex 0040 = Transfer data between main storage and auxiliary in blocks of 32K.
			Hex 0080 = Transfer data between main storage and auxiliary in blocks of 64K.
			Hex 0100 = Transfer data between main storage and auxiliary in blocks of 128K.
74	4A		Allow exchange operations Char(1
			Hex C5 = Allow exchange operations
			Hex D5 = Disable exchange operations
			Hex D9 = Indicate that objects are good candidates for replacement
75	4B		Handling of requests to transfer object to auxiliary storage Char(1
			Hex D5 = Use the system page replacement algorithm
			Hex D7 = Purge the objects from main storage
			Hex D9 = Indicate the objects are good candidates for replacement
			Hex E6 = Write the objects to auxiliary storage
76	4C		Reserved (binary 0) Char(4
104	68		Reserved (binary 0) Char(32)
*	*	—- End —-	

Current number of pools is a user-specified value for the number of storage pools the user wishes to utilize. These are assumed to be numbered from 1 to the number specified. This number is fixed by the machine to be equal to the maximum number of pools.

Type of pool tuning determines what the system is doing to tune the performance of a storage pool.

When *no tuning is being done for a pool* (hex 00), the system tries to minimize the amount of main storage that is used by each of the jobs in the system independent of the amount of main storage that exists in a pool. The values returned for nondatabase objects and database objects by class will be all zeros to represent that the default values are being used.

If *static tuning* is being done (hex 10), the system will use the values specified for pool information to determine the amount of data to transfer to main storage and auxiliary storage.

When *dynamic tuning of transfers to main storage* is being done (hex 20), the system bases the amount of data to transfer to main storage based on the demand for storage in the storage pool, the size of the pool, the number of active users in the pool and other performance attributes. The values returned for database objects by class and nondatabase objects is the current value being used by the system to handle the objects.

When *dynamic tuning of transfers to main storage and auxiliary storage* is being done (hex 30), the system bases the amount of data to transfer to main storage and to auxiliary storage based on the demand for storage in the storage pool, the size of the pool, the number of active users in the pool and other performance attributes. The values returned for database objects by class and nondatabase objects is the current value being used by the system to handle the objects.

When tuning is requested (hex 10, 20 or 30), the system periodically categorizes database objects into four different performance classes. The classes are:

Class 1	Object access appears to be very random - a disk access is required for nearly each record that is accessed
Class 2	Some locality of reference detected, several records are being accessed per disk access
Class 3	High locality of reference detected, object is being processed in a sequential manner, references are highly clustered, large portions of the object are resident in memory.
Class 4	See following explanation.

The class of a database object is adjusted if the object's size is small in comparison to the available storage in the storage pool. This class adjustment involves adding 1 to the class number, so a class 3 database object (as defined above) would be treated as a class 4 if it is small in comparison to the available storage in the storage pool.

Reference information for determining an object's class is collected periodically and by storage pool so an object's class will vary over time and by storage pool.

Changed page handling affects when the system will write changed pages to auxiliary storage. When the *system page replacement algorithm* (hex 00) is specified as the changed page handling mechanism, the system will transfer changed pages to auxiliary storage when:

-
- Explicitly requested to transfer the page (for example, Set Access State (SETACST) instruction)
- There is a demand for pages in the pool

When the *periodically transfer changed pages* option (hex 10) is specified as the changed page handling mechanism, the system will transfer changed pages to auxiliary storage when:

•
- Explicitly requested to transfer the page (for example, Set Access State (SETACST) instruction)
- There is a demand for pages in the pool
- Periodically look for changed pages in a pool and transfer the changed pages to auxiliary storage

Blocking factor determines how much data should be brought into main storage when the object is needed in main storage.

Allow exchange operations controls which method the system should use to find main storage to hold data. With the *exchange method* (hex C5), the system uses the page frames associated with a specific object to satisfy the request. If *exchange operations are disabled* (hex D5), the system will use the normal page replacement algorithm to find page frames for the request. If *objects should be treated as good candidates for replacement* (hex D9), the system makes the page frames associated with the object being exchanged a good replacement candidate but uses the normal page replacement algorithm to find page frames for the request.

Handling of requests to transfer object to auxiliary storage determines when the data is transferred to auxiliary storage and when the page frames containing the object are available to contain other data. If *purging is active* (hex D7) and a request is made to purge the object to auxiliary storage, the system will immediately schedule the request to transfer the data and when the transfer is completed, the page frames containing the data just written will be made available to hold other objects. If *writing is active* (hex E6) and a request is made to purge the object to auxiliary storage, the system will immediately schedule the request to transfer the data and the page frames are not made good candidates to be reused. If *objects are good candidates for replacement* (hex D9), the objects are likely to be removed from main storage by transferring the objects to auxiliary storage when the system needs to transfer other objects into main storage. If the *system page replacement algorithm* is used (hex D5), the system decides when the object should be transferred from main storage to auxiliary storage.

Delay cost scheduling information (Hex 15):

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Scheduling type	Bin(2)
		Hex 0000 = Indicates that delay cost scheduling is disabled. Processes are dispatched on the basis of their assigned priority.	
		Hex 0001 = Indicates that delay cost scheduling is enabled. The machine default priority mapping and cost curve definitions are being used.	
18	12	—- End —-	

MPL Control Data (Hex 16):

Note: Option hex 16 is the preferred method of materializing MPL control information.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Machine-wide MPL control	Char(20)	
16	10		Machine maximum number of MPL classes	Bin(2)
18	12		Machine current number of MPL classes	Bin(2)
20	14		MPL (max)	UBin(4)
24	18		Ineligible event threshold	UBin(4)
28	1C		MPL (current)	UBin(4)
32	20		Number of threads in ineligible state	UBin(4)
36	24	MPL class information	[*] Char(32)	
		(repeated for each MPL class, from 1 to the current number of MPL classes)		
36	24		MPL (max)	UBin(4)
40	28		Ineligible event threshold	UBin(4)
44	2C		Current MPL	UBin(4)
48	30		Number of threads in ineligible state	UBin(4)
52	34		Number of threads assigned to class	UBin(4)
56	38		Number of active to ineligible transitions	UBin(4)
60	3C		Number of active to MI wait transitions	UBin(4)
64	40		Number of MI wait to ineligible transitions	UBin(4)
*	*	—- End —-		

Machine-Wide MPL Control:

Maximum number of MPL classes is the largest number of MPL classes allowed in the machine. These are assumed to be numbered from 1 to the maximum.

Machine current number of MPL classes is a user-specified value for the number of MPL classes in use. They are assumed to be numbered from 1 to the current number.

MPL (max) is the maximum number of threads which may concurrently be in the active state in the machine.

Ineligible event threshold is a number which, if exceeded by the *number of threads in ineligible state* defined below, will cause an event to be signaled. When the event is signaled, this value is set by the machine to an implementation defined value which will be materialized as hex FFFFFFFF. This is done to indicate that the threshold has been exceeded and that the event will not be re-signaled unless the threshold is reset.

MPL (current) is the current number of threads in the active state.

Number of threads in the ineligible state is the number of threads not currently active because of enforcement of both the machine and class MPL rules.

MPL Class Information

MPL class information is data in an array that is associated with an MPL class by virtue of its ordinal position within the array.

MPL (max) is the number of threads assigned to the class which may be concurrently active.

Ineligible event threshold, **MPL (current)**, and **number of threads in ineligible state** are as defined above but apply only to threads assigned to the class.

Number of threads assigned to class is the total number of threads, in any state, assigned to the class.

The total number of transitions among the active, wait, and ineligible states by threads assigned to a class are:

1. **Number of active to ineligible transitions**
2. **Number of active to MI wait transitions**
3. **Number of MI wait to ineligible transitions**

Note that transitions from wait state to active state can be derived as (2 - 3) and transitions from ineligible state to active state as (1 + 3). On overflow, the machine wraps these UBin(4) numbers from hex FFFFFFFF to 0 without any indication of error.

Allocation and De-allocation counts per task and thread (Hex 17):

The materialized data should not be used for accounting purposes as the intended use of this data is for diagnostic purposes, such as, to help determine which task or thread is currently consuming large amounts of space on the system. The user of this MI instruction should be aware that process initiation and termination will be slowed by over use.

Note that through appropriate setting of the *number of bytes provided for materialization* field for operand 1, the amount of information to be materialized for this option can be reduced thus avoiding the processing for unneeded information.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Control information	Char(16)	
16	10		Requested function	UBin(2)
			User specified values:	
			1 = Sorted by storage allocation	
			2 = Sorted by storage de-allocation	
			3 = Sorted by delta storage (allocated minus de-allocated)	
18	12		Total number of tasks and threads	UBin(4)
22	16		Total number of entries	UBin(4)
26	1A		Reserved (binary 0)	Char(6)
32	20	Task and thread information	[*] Char(80)	
		(Repeated once for each task or thread. Located immediately after the control information above.)		
32	20		Task and thread control information	Char(2)
32	20		Task and thread indicator	Bits 0-1

Offset Dec	Offset Hex	Field Name	Data Type and Length		
			00 =	Secondary thread	
			01 =	Initial thread	
			10 =	Task	
			11 =	Reserved	
32	20		Reserved (binary 0)		Bits 2-15
34	22		Reserved (binary 0)		Char(2)
36	24		Task name		Char(32)
68	44		Task identifier		Char(4)
72	48		Thread identifier		Char(8)
80	50		Allocated storage		UBin(4)
84	54		De-allocated storage		UBin(4)
88	58		Delta storage		UBin(4)
92	5C		Reserved (binary 0)		Char(20)
*	*	—- End —-			

Requested function is the option on how the data should be returned back to the requester. This field is input from the requester. The sorting is performed on all the data before determining which elements will be returned.

Total number of tasks and threads is the total number of tasks and threads on the system at the time of the sampling. This includes all machine tasks, initial threads and secondary threads.

Total number of entries is the number of *task and thread information* elements that are being returned.

Task and thread information

Task and thread indicator specifies whether the element is for a task, initial thread or a secondary thread.

Task name is the name of the task. All threads within a process will have the same process control space (PCS) name.

Task identifier contains a value assigned by the machine, which uniquely identifies the task within the machine for as long as the task exists.

Thread identifier contains a value assigned by the machine, which uniquely identifies this thread within its process. The value will not be re-assigned to another thread within the process. For a task this field will be zero.

Allocated storage is the amount of auxiliary storage in pages that has been allocated by this task or thread. The value of this field only increases over time.

De-allocated storage is the amount of auxiliary storage in pages that has been de-allocated by this task or thread. The value of this field only increases over time.

Delta storage is the amount of auxiliary storage in pages that is the difference between the amount allocated and de-allocated by this task or thread. If the de-allocated storage is larger than the allocated storage, then the field will be set to zero.

Processor Multi-tasking mode (hex 18):

This option is used to materialize the pending and current values for the processor multi-tasking mode.

Offset Dec	Offset Hex	Field Name		Data Type and Length
16	10	Current mode		UBin(2)
			0 = Processor multi-tasking capability is currently disabled.	
			1 = Processor multi-tasking capability is currently enabled.	
			2 = Processor multi-tasking capability is currently system controlled.	
18	12	Pending mode		UBin(2)
			0 = Processor multi-tasking capability is to be disabled on the next IPL if supported by the system hardware.	
			1 = Processor multi-tasking capability is to be enabled on the next IPL if supported by the system hardware.	
			2 = Processor multi-tasking capability is to be system controlled on the next IPL if supported by the system hardware.	
20	14	—- End —-		

The **current mode** field returns the current value for the processor multi-tasking mode.

The **pending mode** field returns the pending value for the processor multi-tasking mode.

On the next IPL, if the *pending mode* is set to a supported value for the hardware, the *current mode* will be changed to the *pending mode*. However, if the *pending mode* is set to an unsupported value for the hardware, the value specified for *pending mode* will be ignored and the *pending mode* will be reset to the *current mode* on the next IPL. At IPL, the *pending mode* is set to the *current mode* after changes to the *current mode*, if any, have been applied.

For a physical machine with firmware level hex 00:

•

- All partitions take the value for *current mode* and *pending mode* from the primary partition. MATMATR option hex 01E0 can be used to materialize the firmware level. A physical machine IPL is required for the *pending mode* to become the *current mode*. The default value is 1 (processor multitasking enabled) if supported by the system hardware. Otherwise, the default value is 0 (processor multitasking disabled). Both the *current mode* and *pending mode* are set to the default value on the initial IPL.

For a physical machine with firmware level hex 10:

•

- *Current mode* and *pending mode* are materialized for the current partition. A partition IPL is required for the *pending mode* to become the *current mode*. The default value is 2 (processor multitasking is system controlled). Both the *current mode* and *pending mode* are set to the default value on the initial IPL.

Dynamic priority adjustment mode (hex 19):

Offset				
Dec	**Hex**	**Field Name**		**Data Type and Length**
16	10	Current mode		UBin(2)
		0 =	Indicates that dynamic priority adjustment is currently disabled. Tasks will be dispatched on the basis of their assigned priority.	
		1 =	Indicates that dynamic priority adjustment is currently enabled. This algorithm is effective on systems that have throughput rated for both interactive and non-interactive workloads.	
18	12	Pending mode		UBin(2)

Offset Dec	Hex	Field Name	Data Type and Length
		0 = Indicates that dynamic priority adjustment is to be disabled. Tasks will be dispatched on the basis of their assigned priority.	
		1 = Indicates that dynamic priority adjustment is to be enabled. This algorithm is effective on systems that have throughput rated for both interactive and non-interactive workloads.	
20	14	—- End —-	

Changes to the dynamic priority adjustment mode take effect on the subsequent IPL. The default value for the **current mode** field is ″enabled″ even though the capability is not available on all hardware models. Dynamic priority adjustment mode will not be effective if delay cost scheduling (see option hex 15) has been disabled.

The underlying function, Server Dynamic Tuning, allows the interactive workload on the system to be depressed to allow the non-interactive workload more throughput. As interactive tasks utilize more than a predetermined amount of CPU cycles, their priorities will be lowered to allow non-interactive tasks to obtain CPU cycles. As interactive tasks utilize less than a predetermined amount of CPU cycles, their priorities will be raised toward their assigned priority.

Disk collection / balancing status (hex 1A):

Offset Dec	Hex	Field Name	Data Type and Length
16	10	ASP number	UBin(2)
18	12	Type of disk balance	UBin(2)
		0 = No disk balancing	
		1 = Capacity disk balancing	
		2 = Usage disk balancing	
		3 = Archiving disk balancing	
		4 = Clear the collection data	
		5 = Move data balancing	
20	14	Reserved (binary zero)	Char(16)
36	24	ASP disk collection status	Char(2)

Offset Dec	Hex	Field Name	Data Type and Length	
		Hex 0000 = No collection in process		
		Hex 0001 = Collection in process		
		Hex 0002 = Canceling the collection		
		Hex 0003 = Holding the collection data for balancing only		
		Hex 0004 = Clearing the collection data		
		Hex 0005 = Holding the collection data for collection or balancing		
38	26	ASP disk balancing status	Char(2)	
		Hex 0000 = No balancing in process		
		Hex 0001 = Balancing in process		
		Hex 0002 = Balancing has been cancelled		
		Hex 0003 = Balancing has been suspended		
		Hex 0004 = Balancing has completed		
40	28	Date/time the collection last started	Char(17)	
40	28		Year collection last started	Char(4)
44	2C		Month collection last started	Char(2)
46	2E		Day collection last started	Char(2)
48	30		Hour collection last started	Char(2)
50	32		Minute collection last started	Char(2)
52	34		Reserved (binary zero)	Char(5)
57	39	ASP flags	Char(1)	
57	39		ASP contains compressed and non-compressed units	Bit 0
57	39		ASP is varied on	Bit 1
57	39		Remote mirrored independent ASP is partially varied on	Bit 2
57	39		Reserved (binary 0)	Bits 3-7
58	3A	Number of allocated auxiliary storage units in ASP Note: Number of configured, non-mirrored units + number of mirrored pairs	Bin(2)	
60	3C	Cumulative minutes the collection has run	UBin(4)	
64	40	Date/time the collection last ended	Char(17)	
64	40		Year collection last ended	Char(4)
68	44		Month collection last ended	Char(2)
70	46		Day collection last ended	Char(2)
72	48		Hour collection last ended	Char(2)
74	4A		Minute collection last ended	Char(2)

Offset Dec	Hex	Field Name	Data Type and Length	
76	4C		Reserved (binary zero)	Char(5)
81	51	Date/time the balancing last started	Char(17)	
81	51		Year balancing last started	Char(4)
85	55		Month balancing last started	Char(2)
87	57		Day balancing last started	Char(2)
89	59		Hour balancing last started	Char(2)
91	5B		Minute balancing last started	Char(2)
93	5D		Reserved (binary zero)	Char(5)
98	62	Reserved (binary zero)	Char(2)	
100	64	Cumulative minutes the balancing has run	UBin(4)	
104	68	Date/time the balancing last ended	Char(17)	
104	68		Year balancing last ended	Char(4)
108	6C		Month balancing last ended	Char(2)
110	6E		Day balancing last ended	Char(2)
112	70		Hour balancing last ended	Char(2)
114	72		Minute balancing last ended	Char(2)
116	74		Reserved (binary zero)	Char(5)
121	79	Reserved (binary zero)	Char(7)	
128	80	Amount to be moved	Char(8)	
136	88	Amount moved	Char(8)	
144	90	—- End —-		

ASP number is an input value that uniquely identifies the auxiliary storage pool from which the current collection and balancing status is desired. The ASP number may have a value from 1 through 255. A value of 1 indicates the system ASP. A value of 2 through 255 indicates a user ASP.

Type of disk balance identifies the type of balance activity that is currently running or was done last for this ASP.

ASP disk collection status identifies the requested ASP's current collection status.

-
- *No collection in process* (hex 0000) indicates that no collection is active for this ASP. Also no collection data is available for balancing.
- *Collection in process* (hex 0001) indicates a collection is currently in process. The collection data consists of how frequently data is referenced from a disk arm in the ASP and how busy the disk is during the collection period.
- *Canceling the collection* (hex 0002) indicates the collection has been cancelled. The collection can be resumed or can be used for balancing purposes.
- *Holding the collection data for balancing only* (hex 0003) indicates that the collection data is being held for further movement or clearing.
- *Clearing the collection data* (hex 0004) indicates the collection is currently being cleared.
- *Holding the collection data for collection or balancing* (hex 0005) indicates the collection is currently being held for additional collection, balancing or clearing purposes.

ASP disk balancing status identifies the requested ASP's current balancing status.

- *No balancing in process* (hex 0000) indicates that no balancing is active for this ASP.
- *Balancing in process* (hex 0001) indicates a balancing is currently in process. A disk collection must have been run prior to this activity.
- *Balancing has been cancelled* (hex 0002) indicates the balancing is being cancelled. The balancing can be resumed or the collection data can be cleared.
- *Balancing has been suspended* (hex 0003) indicates the balancing was previously stopped. The balancing can be resumed or the collection data can be cleared.
- *Balancing has completed* (hex 0004) indicates the balancing has completed normally.

Date and time the collection last started. This is an EBCDIC date and time representation indicating when the last collection period was started. If no collection has been started, then the field will be binary zeroes.

- **Year collection last started**
- **Month collection last started**
- **Day collection last started**
- **Hour collection last started**
- **Minute collection last started**

ASP contains compressed and non-compressed units flag specifies whether or not the ASP has compressed and non-compressed configured units. A value of binary 1 indicates that both compressed and non-compressed units exist in this ASP. A value of binary 0 indicates that a mix of compressed and non-compressed units does not exist in this ASP.

ASP is online flag specifies if the ASP is available to the system. If this ASP is an independent ASP, a value of binary 1 indicates the independent ASP is varied on. If this ASP is an independent ASP, a value of binary 0 indicates the independent ASP is varied off. A value of binary 1 is returned if the ASP is a basic ASP or a system ASP.

Remote mirrored independent ASP is partially varied on flag specifies that the remote mirrored copy of an independent ASP is partially varied on. If this ASP is an independent ASP, a value of binary 1 indicates that it is the mirror copy in a remotely mirrored independent ASP and that the independent ASP is partially varied on. If this ASP is an independent ASP, a value of binary 0 indicates the independent ASP is varied off. A value of binary 0 is returned if the ASP is a basic ASP or a system ASP.

Number of allocated auxiliary storage units in ASP is the number of configured units logically addressable by the system as units for this ASP. This is the number of configured, non-mirrored units plus the number of mirrored pairs allocated in the ASPs. The total number of units (actuator arms) on the system is the sum of the allocated auxiliary storage units plus the number of unallocated auxiliary storage units plus the number of pairs of mirrored units. For example, each 9335 enclosure represents two units. Information on these units is materialized as part of the unit information. Any two units of the same size may be associated to form a mirrored pair. Association of two units as a mirrored pair reduces the amount of logically available storage by the number of bytes contained on one of the mirrored units in the mirrored pair.

Cumulative minutes the collection has run. Since the collection can be stopped and restarted several times this gives the user an indication of how long the collection has been run. When the collection is cleared this field is reset to binary zeroes.

Date and time the collection last ended. This is an EBCDIC date and time representation indicating when the last collection period was ended. If no collection has ended, then the field will be binary zeroes.

-
- **Year collection last ended**
- **Month collection last ended**
- **Day collection last ended**
- **Hour collection last ended**
- **Minute collection last ended**

Date and time the balancing last started. This is an EBCDIC date and time representation indicating when the last balancing period was started. If no balancing has been started, then the field will be binary zeroes.

-
- **Year balancing last started**
- **Month balancing last started**
- **Day balancing last started**
- **Hour balancing last started**
- **Minute balancing last started**

Cumulative minutes the balancing has run. Since the balancing can be stopped and restarted several times this gives the user an indication of how long the balancing has been run. When the collection is cleared this field is reset to binary zeroes.

Date and time the balancing last ended. This is an EBCDIC date and time representation indicating when the last balancing period was ended. If no balancing has been ended, then the field will be binary zeroes.

-
- **Year balancing last ended**
- **Month balancing last ended**
- **Day balancing last ended**
- **Hour balancing last ended**
- **Minute balancing last ended**

Amount to be moved. This is the target amount in megabytes that the balancing function will attempt to re-balance.

Amount moved. This is the amount in megabytes that the balancing function has moved to re-balance.

Materialize mapping of partition processors (Hex 1B):

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Number of processors for which information is being materialized	UBin(2)
18	12	Reserved (binary 0)	Char(6)
24	18	Physical processor token	[*] UBin(2)
*	*	—- End —-	

Number of processors for which information is being materialized is the number of the virtual processors that are currently active in the partition. *Number of processors for which information is being materialized* is less than or equal to the *number of processors configured on the machine* returned by MATRMD option hex 13.

Physical processor token provides an index that can be used to correlate the virtual processor to its vital product data returned by MATMATR option hex 012C for a system with a maximum of 16 processors. *Physical processor token* is the index of the physical machine processor (starting from 1) that a partition virtual processor is currently mapped to.

For a partition sharing physical processors, this mapping only provides a snapshot. At a given instance, a partition processor may be mapped to any of the physical processors in the shared pool in which the partition is running.

DASD Management Status (Hex 1C):

This option returns status information from the DASD manager.

The format of the template for the status information from the DASD manager follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Reserved (binary 0)	Char(8)
24	18	Handle	Char(8)
32	20	Status	UBin(2)
		0 = DASD Management not in use.	
		1 = DASD Management in use by an MI user. No action in progress.	
		2 = DASD Management in use by an MI user. Action in progress.	
		3 = DASD Management in use by DST/service tools	
34	22	Action identifier	UBin(2)

Offset Dec	Offset Hex	Field Name	Data Type and Length
		0001 = Change ASP event threshold percentage	
		0002 = Add disk units	
		0003 = Suspend mirrored protection	
		0004 = Resume mirrored protection	
		0005 = Include disk units in an existing parity set	
		0006 = Rebuild data on a unit after a parity fault	
		0007 = Replace unit with a non-configured unit	
		0008 = Start device parity protection	
		0009 = Enable remote load source mirroring	
		0010 = Disable remote load source mirroring	
		0011 = Power off unit	
		0012 = Power on unit	
		0013 = Format disk	
		0014 = Blank disk	
		0015 = Surface scan of disk	
		5000 = DST/service tools actions	
		5001 = No actions performed	
		5002 = Specified handle not performing action	
36	24	Percentage complete	UBin(2)
38	26	Number of return codes	UBin(2)
40	28	Return code array	[*] Char(*)
*	*	—- End —-	

The **handle** is an optional input field. If provided, the *handle* will return the status for the open connection to DASD management. If there is not an open connection to DASD management or the *handle* does not match the handle of the open connection, *action identifier* will be set to 5002 and no additional information will be returned. If *handle* is set to hex zeros, all available information about DASD management will be returned, including *status*, *action identifier*, *percentage complete*, *number of return codes*, and *return code array*. The *handle* has a timeout associated with it. If the *handle* is not used to perform an action or is not used to check the status using this MATRMD option within 5 minutes after an action completes, the connection to DASD Management is automatically closed and the *handle* is invalidated and cannot be used for any more DASD management actions. This timeout counter only starts when an action is completed. For example, if a long running action such as *add disk units* takes an hour to complete, the timer will be started after the *add disk units* action completes. The connection will close 5 minutes after the *add disk units* has completed if the *handle* has not been used to request another action or check the status within that 5 minutes. Once the *handle* is used to check the status or request a new action, the timer is reset.

The **status** field is an output field which specifies the status of DASD Management.

The **action identifier** field is an output field which specifies the most recently attempted action if no action is in progress or the action in progress. If the action specified is 5001 (no actions performed), or 5002 (DST/service tools action), the *status* field is the only other valid output field.

The **percentage complete** field is an output field which identifies the percentage complete of an action being performed for an MI user. This field is not defined if DASD Management is currently being used by DST/service tools or if no action is currently being performed. The range of percentages that may be returned is 0 through 100.

The **number of return codes** is an output field that identifies the number of return codes in the *return code array*. If the *number of return codes* is zero and the *percentage complete* is 100, the action completed successfully.

The **return code array** is an output field that refers to an array of return codes and data associated with the *action identifier*. If there is not enough space allocated for all return codes in the array, no return code information will be filled in.

The format of an entry in the return code array follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Return code	UBin(2)
		Hex 0000 = Action successful	
		Hex 0004 = Action failed	
		Hex 00FF = General DASD management error	
		Hex 0302 = Cannot restore mirrored data	
		Hex 0402 = Disk unit has errors	
		Hex 0600 = Create new ASP failed	
		Hex 0706 = Cannot rebuild parity information	
		Hex 0708 = Device parity set not operational	
		Hex 0902 = Action was cancelled	
2	2	Return code details	Char(30)
32	20	—- End —-	

The **return code** field is an output field which identifies one of the return codes of the most recently completed DASD management action.

The **return code details** field is an output field which identifies error data associated with the *return code*. If the failure involves a disk unit, the resource name of that disk unit will be placed in the first ten characters of this field. The remaining 20 characters of this field are reserved for future use. If the failure does not involve a disk unit, this field will be set to binary zeros and should be ignored.

DASD Management Disk Information (hex 1D):

This option provides information about a list of disks.

The format of the template for disk information follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Reserved (binary 0)	Char(8)
24	18	Number of elements in disk information array	UBin(4)
28	1C	Length of element in disk information array	UBin(4)
32	20	Disk information array	[*] Char(*)
*	*	—- End —-	

Note: This template must be 16 byte aligned.

The **number of elements in disk information array** field is an input/output field which specifies the number of elements in the *disk information array*.

The **length of element in disk information array** field is an input field which specifies the length, in bytes, of an entry in the *disk information array* field.

The **disk information array** is an array of disk resource names and information about those disks. The format of the *disk information array* is as follows:

Note: The length of the *disk information array* is *number of elements in disk information array * length of element in disk information array*. If the actual length of disk information array is smaller than this value, the *number of elements in disk information array* field will be updated to the number of elements provided.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Disk resource name	Char(10)	
10	A	Flags	Char(1)	
10	A		Disk unit may be included in new parity set	Bit 0
			0 = Not allowed in new parity set	
			1 = Allowed in new parity set	
10	A		Disk unit may be included in existing parity set	Bit 1
			0 = Not allowed in existing parity set	
			1 = Allowed in existing parity set	
10	A		Disk unit not found	Bit 2
			0 = Disk unit found	
			1 = Disk unit not found	
10	A		Reserved (binary 0)	Bits 3-7
11	B	Parity set number	Char(1)	
12	C	Capacity available after parity started	UBin(4)	
16	10	Frame associated with disk unit	Char(10)	
26	1A	Frame ID associated with disk unit	Char(4)	
30	1E	Reserved	Char(2)	
32	20	—- End —-		

The **disk resource name** field is an input field which specifies the name of the disk unit to return information about. The disk unit resource name of the first element in the array may have the special value of '*UNCONFIG' which indicates all unconfigured disk units in the system will be found and the associated parity information for those disks will be returned. The *number of elements in disk information array* field will be updated to the number of elements provided.

The **disk unit may be included in a new parity set** field is an output field which specifies if the *disk resource name* can be included as one of the disks in that set when creating a new parity set.

The **disk unit may be included in existing parity set** field is an output field which specifies if *disk resource name* is eligible to be on a list of disks that is to be added to that parity set.

The **disk unit not found** field is set to a 1 if a disk unit corresponding to the *disk resource name* was not found on the system.

The **parity set number** field is an output field that specifies the parity set a disk unit will belong to after it has been included in a parity set. The value of this field should be ignored if the disk unit is not allowed in a new or existing parity set.

The **capacity available after parity started** field is an output field that specifies the capacity of the disk unit in millions of bytes after this disk unit becomes part of a parity set. The value of this field should be ignored if the disk unit is not allowed in a new or existing parity set.

The **frame associated with disk unit** field is an output field that identifies the frame resource to which the disk unit is attached. This field may be used to determine the physical location of the disk unit.

The **frame ID associated with disk unit** field is an output field that identifies the frame id to which the disk unit is attached. This field may be used to determine the physical location of the disk unit.

Interactive Utilization Data (Hex 1E):

This option provides information about interactive utilization. For additional information, see manual SC41-0607 iSeries Performance Capabilities Reference manual which is available in the iSeries Information Center.

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
16	10	Interactive threshold	UBin(2)
18	12	Interactive limit	UBin(2)
20	14	Reserved	Char(4)
24	18	Interactive processor usage since IPL	Char(8)
32	20	Interactive processor usage above threshold since IPL	Char(8)
40	28	Reserved	Char(16)
56	38	—- End —-	

Interactive threshold is the highest level of interactive processor utilization which can be sustained without causing a disproportionate increase in system overhead. The value returned is the fraction of processor capacity, expressed in tenths of a percent. For example, a value of 237 means that the threshold is 23.7%.

On a machine with no limit on interactive utilization, the value returned will be 1000 (100%).

Interactive limit is the maximum sustainable level of interactive processor utilization. The machine determines the *interactive limit* based on the *interactive feature*. The value returned is the fraction of processor capacity, expressed in tenths of a percent. For example, a value of 275 means that the limit is 27.5%.

On a machine with no limit on interactive utilization, the value returned will be 1000 (100%).

Interactive processor usage since IPL is the total processor time, used by interactive processes since IPL. If the system does not support this metric, a value of hex 0000000000000000 is returned. If the system

does support this and needs to return a value of 0, a value of hex 0000000000001000 is returned. For all other cases, the significance of bits within this field is the same as that defined for the time-of-day clock. On a machine with more than one virtual processor, the value returned will be the sum of the *interactive processor usage since IPL* for all virtual processors.

Interactive processor usage above threshold since IPL is the total processor time, used by interactive processes, since IPL, during which the interactive utilization exceeded the *interactive threshold*. On a machine with more than one virtual processor, the value returned will be the sum of the *interactive processor usage above threshold since IPL* for all virtual processors. The significance of bits within this field is the same as that defined for the time-of-day clock.

For a partition using shared processors, *interactive processor usage since IPL* and *interactive processor usage above threshold since IPL* are scaled by the *configured capacity* of the partition. This allows CPU utilization calculations to be done as if the partition was using whole physical processors.

Auxiliary Storage Pool Information (Short format) (Hex 1F):

The *auxiliary storage pool information* describes the ASPs (auxiliary storage pools) which are configured within the machine. This option does not return information for independent ASPs which are varied off. You can use option "Auxiliary Storage Pool Information including offline Independent ASPs (Hex 22)" (page 899) to return information about independent ASPs which are varied off.

Also note that through appropriate setting of the *number of bytes provided* field for operand 1, the amount of information to be materialized for this option can be reduced thus avoiding the processing for unneeded information.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Control information (occurs just once)	Char(16)	
16	10		Number of ASPs	UBin(2)
18	12		Reserved (binary 0)	Char(14)
32	20	ASP information (Repeated once for each ASP. Located immediately after the control information above. ASP 1, always configured, is first. Configured ASPs follow in ascending numerical order.)	[*] Char(32)	
32	20		ASP number	Char(2)
34	22		Number of allocated auxiliary storage units in ASP Note: Number of configured, non-mirrored disk units + number of mirrored pairs of disk units	UBin(2)
36	24		ASP resource name	Char(10)
46	2E		ASP control flags	Char(2)
46	2E		ASP overflow	Bi
46	2E		Independent ASP	Bi
46	2E		ASP protected	Bi
46	2E		User ASP MI state	Bi
46	2E		Independent ASP address threshold exceeded	Bi
46	2E		Reserved (binary 0)	Bi
48	30		Number of addresses remaining in independent ASP	Char(8)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
56	38		ASP number of the primary ASP	Char(2)
58	3A		Independent ASP type	Char(1)
58	3A		Primary ASP	Bit 0
58	3A		Secondary ASP	Bit 1
58	3A		UDFS ASP	Bit 2
58	3A		Reserved (binary 0)	Bits 3
59	3B		Reserved (binary 0)	Char(5)
*	*	—- End —-		

Number of ASPs is the number of ASPs configured within the machine. One, the minimum value, indicates just the system ASP exists and that there are no user ASPs configured. Up to 255 user ASPs can be configured. The system ASP always exists. This number of ASPs include the system ASP, basic ASPs (that is, user ASPs which cannot be varied on), and independent ASPs which are currently varied on to this system.

ASP information is repeated once for each ASP configured within the machine. The number of ASPs configured is specified by the *number of ASPs* field. ASP 1, the system ASP, is materialized first. Because the system ASP always exists, its materialization is always available. The information about the user ASPs is materialized after the system ASP in ascending numerical order. There may be gaps in the numerical order. For example, if user ASPs 3 and 75 are configured, the materialize will produce information on ASP 1, ASP 3, and ASP 75 in that order.

ASP number uniquely identifies the auxiliary storage pool. The ASP number may have a value from 1 through 255. A value of 1 indicates the system ASP. A value of 2 through 255 indicates a user ASP. Note that independent ASPs have a value of 33 through 255.

Number of allocated auxiliary storage units in ASP is the number of configured units logically addressable by the system as units for this ASP. This is the number of configured, non-mirrored units plus the number of mirrored pairs allocated in the ASPs. Any two units of the same capacity may be associated to form a mirrored pair. Association of two units as a mirrored pair reduces the amount of logically available storage by the number of bytes contained on one of the mirrored units in the mirrored pair.

ASP resource name specifies the name which the user has assigned to this auxiliary storage pool. Blanks (hex value 40) are returned for ASPs which do not have names. Only independent ASPs have names. The ASP name is the *resource name* in the LUD.

ASP overflow flag indicates whether or not object allocations directed into the basic ASP have overflowed into the system ASP. A value of binary 1 indicates overflow; binary 0 indicates no overflow. This flag does not apply to the system ASP and a value of binary 0 is always returned for it. This flag does not apply to independent ASPs and a value of binary 0 is always returned for independent ASPs.

Independent ASP specifies whether or not the ASP is an independent ASP; that is, a user ASP than can be varied on or off. A value of binary 1 indicates the ASP is an independent ASP. A value of binary 0 indicates that this ASP is a basic ASP (a user ASP that cannot be varied on or off).

ASP protected specifies whether or not the ASP is configured to be protected from a single disk failure. A value of binary 1 indicates that the ASP is protected. All of the disk units in this ASP must be either device parity protected or mirror protected. A value of binary 0 indicates that the disk units in the ASP are not mirror protected, and there is no requirement that the disk units in the ASP be device parity protected.

User ASP MI state indicates the state of the user ASP. A value of binary 1 indicates that the user ASP is in the 'new' state. This means that a context may be allocated in this user ASP. A value of binary 0 indicates that the user ASP is in the 'old' state. This means that there are no contexts allocated in this user ASP. This flag has no meaning for the system ASP and a value of binary 0 will always be returned for the system ASP. A value of binary 1 is always returned for independent ASPs.

Independent ASP address threshold exceeded flag is only valid for an Independent ASP and specifies whether or not the independent ASP address threshold, selected by the machine, has been exceeded. A value of binary 1 indicates the threshold has been exceeded and the Independent ASP is running low on addresses. A value of binary 0 indicates that the address threshold has not been exceeded.

Number of addresses remaining in independent ASP contains the number of virtual addresses remaining for use by the independent ASP. This field only has meaning for an independent ASP. The information in this field is only valid if the *independent ASP address threshold exceeded* flag is set to binary 1.

ASP number of the primary ASP contains the ASP number of the primary ASP. This value only has meaning for an independent ASP. If the ASP is a secondary ASP, this field contains the ASP number of the primary ASP. If the ASP is a primary ASP, this value is the same as the *ASP number*. If the ASP is a UDFS ASP or is not an independent ASP, a value of hex 0000 is returned.

Primary ASP flag indicates that the independent ASP is a primary ASP in an ASP group. A primary ASP defines a collection of directories and contexts and may have secondary ASPs associated with it. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a primary ASP. A value of binary 0 indicates the independent ASP is not a primary ASP.

Secondary ASP flag indicates that the independent ASP is a secondary ASP in an ASP group. A secondary ASP is associated with a primary ASP. There can be many secondary ASPs associated with the same primary ASP. The secondary ASP defines a collection of directories and contexts. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a secondary ASP. A value of binary 0 indicates the independent ASP is not a secondary ASP.

UDFS ASP flag indicates that the independent ASP is a UDFS (User-defined File System) ASP. This type of independent ASP cannot be a member of an ASP group. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a UDFS ASP. A value of binary 0 indicates the independent ASP is not a UDFS ASP.

Auxiliary Storage information including offline Independent ASPs (Hex 20):

The *auxiliary storage information* describes the ASPs (auxiliary storage pools) which are configured within the machine and the units of auxiliary storage currently allocated to an ASP or known to the machine but not allocated to an ASP. This option returns information for all ASPs including independent ASPs that are varied off. Option "Auxiliary Storage Information (Hex 12)" (page 843) returns the same information but does not return information for independent ASPs that are varied off.

Also note that through appropriate setting of the number of bytes provided field for operand 1, the amount of information to be materialized for this option can be reduced thus avoiding the processing for unneeded information. As an example, by setting this field to only provide enough bytes for the common 16 byte header, plus the option hex 20 control information, plus the system ASP entry of the ASP information, you can get just the information up through the system ASP entry returned and avoid the overhead for the user ASPs and unit information.

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
16	10	Control information (occurs just once)	Char(64)

Offset Dec	Hex	Field Name	Data Type and Length	
16	10		Number of ASPs Note: 1 (the system ASP) + number of basic ASPs + number of varied-on independent ASPs + number of varied off independent ASPs	Bin(2)
18	12		Number of allocated auxiliary storage units Note: Number of configured, non-mirrored units + number of mirrored pairs	Bin(2)
20	14		Number of unallocated auxiliary storage units	Bin(2)
22	16		Reserved (binary 0)	Char(2)
24	18		Maximum auxiliary storage allocated to temporaries	Char(8)
32	20		Reserved (binary 0)	Char(12)
44	2C		Unit information offset	Bin(4)
48	30		Number of pairs of mirrored units	Bin(2)
50	32		Mirroring main storage	Bin(4)
54	36		Number of multipath units	UBin(2)
56	38		Current auxiliary storage allocated to temporaries	Char(8)
64	40		Number of bytes in a page	Bin(4)
68	44		Number of independent ASPs	UBin(2)
70	46		Number of disk units in all independent ASPs	UBin(2)
72	48		Number of basic ASPs	UBin(2)
74	4A		Number of disk units in all basic ASPs	UBin(2)
76	4C		Number of disk units in the system ASP	UBin(2)
78	4E		Number of additional entries for multipath units	UBin(2)
80	50	ASP information (Repeated once for each ASP. Located immediately after the control information above. ASP 1, always configured, is first. Configured user ASPs follow in ascending numerical order.)	[*] Char(160)	
80	50		ASP number	Char(2)
82	52		ASP control flags	Char(1)
82	52		Suppress threshold exceeded event	Bit 0
82	52		ASP overflow	Bit 1
82	52		Reserved	Bits 2-3
82	52		ASP mirrored	Bit 4
82	52		User ASP MI state	Bit 5
82	52		ASP overflow storage available	Bit 6
82	52		Suppress available storage lower limit reached event	Bit 7
83	53		ASP overflow recovery result	Char(1)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
83	53		Successful	Bit 0
83	53		Failed due to insufficient free space	Bit 1
83	53		Cancelled	Bit 2
83	53		Reserved (binary 0)	Bits 3-7
84	54		Number of allocated auxiliary storage units in ASP	UBin(2)
			Note: Number of configured, non-mirrored units + number of mirrored pairs. It is possible that this number is 0 (zero) for an offline Independent ASP	
86	56		Remote mirror performance mode	Char(1)
			Hex 01 = Synchronous mode	
			Hex 02 = Asynchronous mode	
87	57		Remote mirror copy data state	Char(1)
			Hex 00 = Remote IASP mirroring is not configured	
			Hex 01 = Remote copy is in sync with the production copy	
			Hex 02 = Remote copy contains useable data	
			Hex 03 = Remote copy data cannot be used	
88	58		ASP media capacity	Char(8)
96	60		Reserved	Char(8)
104	68		ASP space available	Char(8)
112	70		ASP event threshold	Char(8)
120	78		ASP event threshold percentage	Bin(2)
122	7A		Additional ASP control flags	Char(2)
122	7A		Terminate immediately when out of storage	Bit 0
122	7A		ASP contains compressed and non-compressed units	Bit 1
122	7A		Recover overflowed basic ASP during normal mode IPL	Bit 2
122	7A		Independent ASP	Bit 3
122	7A		ASP is online	Bit 4
122	7A		Independent ASP address threshold exceeded	Bit 5
122	7A		Independent ASP is remote mirrored	Bit 6
122	7A		Reserved (binary 0)	Bits 7-15
124	7C		ASP compression recovery policy	Char(1)
124	7C		Error recovery policy	Bits 0-1

Offset Dec	Hex	Field Name	Data Type and Length	
			00 = Retry while space available **01 =** Overflow immediately **10 =** Retry forever	
124	7C		Reserved (binary 0)	Bits 2-7
125	7D		Independent ASP type	Char(1)
125	7D		Primary ASP	Bit 0
125	7D		Secondary ASP	Bit 1
125	7D		UDFS ASP	Bit 2
125	7D		Reserved (binary 0)	Bits 3-7
126	7E		Remote mirror role	Char(1)
			Hex 00 = Remote IASP mirroring is not configured **Hex 01 =** System does not own a physical independent ASP copy **Hex 02 =** Remote mirror role is unknown **Hex C4 =** System owns a detached mirror copy **Hex D4 =** System owns the mirror copy **Hex D7 =** System owns the production copy	
127	7F		Remote mirror copy state	Char(1)
			Hex 00 = Remote IASP mirroring is not configured **Hex 01 =** System attempts to perform independent ASP remote mirroring when independent ASP is online. **Hex 02 =** Remote independent ASP role is resuming. **Hex 03 =** System is resuming and independent ASP is online and performing synchronization **Hex 04 =** Remote independent ASP is detached and remote mirroring is not being performed.	
128	80		ASP system storage	Char(8)
136	88		ASP overflow storage	Char(8)
144	90		Space allocated to the error log	Bin(4)
148	94		Space allocated to the machine log	Bin(4)
152	98		Space allocated to the machine trace	Bin(4)
156	9C		Space allocated for main store dump	Bin(4)
160	A0		Space allocated to the microcode	Bin(4)
164	A4		Remote mirror synchronization priority	Char(1)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			Hex 00 = Remote IASP mirroring is not configured	
			Hex 10 = Synchronization is given high priority	
			Hex 20 = Synchronization is given medium priority	
			Hex 30 = Synchronization is given low priority	
165	A5		Remote mirror encryption mode	Char(1)
			Hex 00 = Remote IASP mirroring is not configured	
			Hex 01 = Data being sent to remote mirror site is not encrypted	
			Hex 012= Data being sent to remote mirror site is encrypted	
166	A6		Remote mirror error recovery	Char(1)
			Hex 00 = Remote IASP mirroring is not configured	
			Hex 02 = Remote mirroring is suspended when an independent ASP error is detected.	
			Hex 03 = Remote mirroring is ended when an independent ASP error is detected.	
167	A7		Remote mirror minutes until timeout	Char(1)
168	A8		Available storage lower limit	Char(8)
176	B0		Protected space capacity	Char(8)
184	B8		Unprotected space capacity	Char(8)
192	C0		Protected space available	Char(8)
200	C8		Unprotected space available	Char(8)
208	D0		Reserved (binary 0)	Char(8)
216	D8		Number of addresses remaining in independent ASP	Char(8)
224	E0		Reserved	Char(16)
*	*	Unit information (Consists of one entry each for the configured, non-mirrored units and one unit of the mirrored pairs, the non-configured units, and the other unit of the mirrored pairs, and an entry for each multipath connection. An allocated storage unit (ASU) is either an allocated, non-mirrored unit or a mirrored pair. Note that the mirrored pair counts only as one ASU. When used without qualification, the term unit refers to an ASU. Unit information start may be located by the Unit Information Offset in the control information.)	[*] Char(208)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
*	*		Device type	Char(8)
*	*		Disk type	Char(4)
*	*		Disk model	Char(4)
*	*		Device identification	Char(8)
*	*		Unit number	Char(2)
*	*		Reserved	Char(6)
*	*		Reserved	Char(4)
*	*		Unit ASP number	Char(2)
*	*		Logical mirrored pair status	Char(1)
*	*		Unit mirrored	Bit 0
*	*		Mirrored unit protected	Bit 1
*	*		Mirrored pair reported	Bit 2
*	*		Reserved	Bits 3-7
*	*		Mirrored unit status	Char(1)
*	*		Unit media capacity	Char(8)
*	*		Unit storage capacity	Char(8)
*	*		Unit space available	Char(8)
*	*		Unit space reserved for system	Char(8)
*	*		Reserved	Char(6)
*	*		Unit control flags	Char(2)
*	*		Reserved (binary 0)	Bit 0
*	*		Unit is device parity protected	Bit 1
*	*		Subsystem is active	Bit 2
*	*		Unit in subsystem has failed	Bit 3
*	*		Other unit in subsystem has failed	Bit 4
*	*		Subsystem runs in degraded mode	Bit 5
*	*		Hardware failure	Bit 6
*	*		Device parity protection is being rebuilt	Bit 7
*	*		Unit is not ready	Bit 8
*	*		Unit is write protected	Bit 9
*	*		Unit is busy	Bit 10
*	*		Unit is not operational	Bit 11
*	*		Status is not recognizable	Bit 12
*	*		Status is not available	Bit 13
*	*		Unit is read/write protected	Bit 14
*	*		Unit is compressed	Bit 15
			Bits 2 to 14 are mutually exclusive.	
*	*		Additional unit control flags	Char(2)
*	*		Do not allocate additional storage on this disk unit	Bit 0
*	*		Unit is in availability parity set	Bit 1
*	*		Unit is multipath unit	Bit 2
*	*		Reserved (binary 0)	Bits 3-15
*	*		Reserved (binary 0)	Char(14)
*	*		Reserved (binary 0)	Char(42)
*	*		Unit Identification	Char(22)
*	*		Serial number	Char(10)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
*	*		Resource name	Char(10)
*	*		Reserved (binary 0)	Char(2)
*	*		Unit usage information	Char(64)
*	*		Blocks transferred to main storage	Bin(4)
*	*		Blocks transferred from main storage	Bin(4)
*	*		Requests for data transfer to main storage	Bin(4)
*	*		Requests for data transfer from main storage	Bin(4)
*	*		Permanent blocks transferred from main storage	Bin(4)
*	*		Requests for permanent data transfer from main storage	Bin(4)
*	*		Reserved (binary 0)	Char(8)
*	*		Sample count	Bin(4)
*	*		Not busy count	Bin(4)
*	*		Reserved (binary 0)	Char(24)
*	*	—- End —-		

Number of ASPs is the number of ASPs configured within the machine. One, the minimum value, indicates just the system ASP exists and that there are no user ASPs configured. Up to 254 user ASPs can be configured. The system ASP always exists. The *number of ASPs* includes the system ASP, basic ASPs (that is, user ASPs which cannot be varied on or off), and independent ASPs. The independent ASPs can be varied on or off on this system, and varied-off (offline) independent ASPs are counted.

Number of allocated auxiliary storage units is the total number of configured units logically addressable by the system as units. This is the number of configured, non-mirrored units plus the number of mirrored pairs allocated to the ASPs. This number includes only the first path of a multipath connection unit. The count of the remaining paths connected to multipath units is materialized in **number of additional entries for multipath units**. The total number of disk actuator arms on the system is the sum of the allocated auxiliary storage units plus the number of unallocated auxiliary storage units plus the number of pairs of mirrored units. Information on these units is materialized as part of the unit information. Any two units of the same size may be associated to form a mirrored pair. Association of two units as a mirrored pair reduces the amount of logically available storage by the number of bytes contained on one of the mirrored units in the mirrored pair. The disk units reside in the system ASP, a basic ASP, or an independent ASP. This number specifies the number of entries which are materialized in the *unit information* section.

Number of unallocated auxiliary storage units is the number of auxiliary storage units that are currently not allocated to an ASP. Information on these units is materialized as part of the unit information.

Maximum auxiliary storage allocated to temporaries is the maximum number of bytes of temporary storage allocated at any one time since the last IPL of the machine. This includes the temporary storage allocated on the load source unit.

Unit information offset is the offset, in bytes, from the start of the operand 1 materialization template to the start of the unit information. This value can be added to a space pointer addressing the start of operand 1 to address the start of the unit information.

Number of pairs of mirrored units represents the number of mirrored pairs in the system. Each mirrored pair consists of two mirrored units; however, only one of the two mirrored units is guaranteed to be operational.

Mirroring main storage is the number of bytes of main storage in the machine storage pool used by mirroring. This increases when mirror synchronization is active. This amount of storage is directly related to the number of mirrored pairs.

Number of multipath units is the number of disk units that have multiple connections to a disk unit. This means that there are multiple resource names that all represent the same disk unit, yet each represents a unique path to the disk unit. All active connections will be used for communicating with the disk unit.

Current auxiliary storage allocated to temporaries is the number of bytes of temporary storage allocated on the system. This includes the temporary storage allocated on the load source unit.

Number of bytes in a page is the number of bytes in a single page. This can be used to convert fields that are given in pages into the correct number of bytes.

Number of independent ASPs is the number of independent ASPs known by this system. An independent ASP is an ASP that can be varied on or off. This count includes independent ASPs which are varied on and varied off.

Number of disk units in all independent ASPs is the number of configured units logically addressable by all independent ASPs. Information on these units is materialized as part of the unit information.

Number of basic ASPs is the number of basic ASPs configured on this system. A basic ASP is a user ASP that cannot be varied on or off.

Number of disk units in all basic ASPs is the total number of configured units logically addressable by all basic ASPs. Information on these units is materialized as part of the unit information.

Number of disk units in the system ASP is the total number of configured units logically addressable in the system ASP. Information on these units is materialized as part of the unit information.

Number of additional entries for multipath units is the number of additional unit entries that can be materialized for the multipath connection devices. The first path of each unit is not included in this total.

ASP information is repeated once for each ASP configured within the machine. The number of ASPs configured is specified by the *number of ASPs* field. ASP 1, the system ASP, is materialized first. Because the system ASP always exists, its materialization is always available. The user ASPs which are configured are materialized after the system ASP in ascending numerical order. There may be gaps in the numerical order. That is, if just user ASPs 3 and 5 are configured, only information for them is materialized producing information on just ASP 1, ASP 3 and ASP 5 in that order.

ASP number uniquely identifies the auxiliary storage pool. The ASP number may have a value from 1 through 255. A value of 1 indicates the system ASP. A value of 2 through 255 indicates a user ASP. Note that independent ASPs have a value of 33 through 255. Note that basic ASPs have a value of 2 through 32.

Suppress threshold exceeded event flag indicates whether or not the machine is suppressing signaling of the related event when the event threshold in effect for this ASP has been exceeded. A value of binary 1 indicates that the signaling is being suppressed; binary 0 indicates that the signaling is not being suppressed. The default after each IPL of the machine is that the signaling is not suppressed; i.e. default is binary 0. For the system ASP, this flag is implicitly set to binary 1 by the machine when the *machine auxiliary storage threshold exceeded* (hex 000C,02,01) event is signaled. For a basic ASP, this flag is implicitly set to binary 1 by the machine when the *user auxiliary storage threshold exceeded* (hex 000C,02,02) event is signaled. If the ASP is an independent ASP and the **ASP is online** flag indicates that the independent ASP is not online, a value of binary 0 is returned for the **suppress threshold exceeded event** flag.

The **ASP overflow** flag indicates whether or not object allocations directed into a basic ASP have overflowed into the system ASP. A value of binary 1 indicates overflow; binary 0 indicates no overflow. This flag does not apply to the system ASP and a value of binary 0 is always returned for it. A value of binary 0 is always returned for independent ASPs.

ASP mirrored flag specifies whether or not the ASP is configured to be mirror protected. A value of binary 1 indicates that ASP mirror protection is configured. Refer to the *mirrored unit protected* flag to determine if mirror protection is active for each mirrored pair. A value of binary 0 indicates that none of the units associated with the ASP are mirrored.

User ASP MI state indicates the state of the user ASP. A value of binary 1 indicates that the user ASP is in the 'new' state. This means that a context may be allocated in this user ASP. A value of binary 0 indicates that the user ASP is in the 'old' state. This means that there are no contexts allocated in this user ASP. This flag has no meaning for the system ASP and a value of binary 0 will always be returned. A value of binary 1 will always be returned for independent ASPs.

ASP overflow storage available flag indicates whether or not the amount of auxiliary storage that has overflowed from the basic ASP into the system ASP is available. A value of binary 1 indicates that the amount is maintained by the machine and available in the *ASP overflow storage* field. A value of binary 0 indicates that the amount is not available. A value of binary 0 is always returned for independent ASPs.

Suppress available storage lower limit reached event flag indicates whether the machine will signal the related event when the available storage lower limit has been reached. This field currently has meaning only in the system ASP (ASP 1). This value will always be returned as binary 0 for a user ASP. A value of binary 1 indicates that signaling of the event is being suppressed; binary 0 indicates that signaling of the event is not suppressed. The default after each IPL of the machine is binary 0, i.e., signaling of this event is not suppressed. This flag is set to binary 1 by the machine when the *available storage lower limit reached* (hex 000C,02,08) event is signaled. This is done to avoid repetitive signaling of the event when the available storage lower limit reached condition occurs.

ASP overflow recovery result flags indicate the result of the ASP overflow recovery operation which is performed during an IPL upon request by the user. When this operation is requested, the machine attempts to recover a basic ASP from an overflow condition by moving overflowed auxiliary storage from the system ASP back to the basic ASP during the Storage Management recovery step of an IPL. The *successful* flag is set to a value of binary 1 when all the overflowed storage was successfully moved. The *failed due to insufficient free space* flag is set to a value of binary 1 when there is not sufficient free space in the basic ASP to move all the overflowed storage. The *cancelled* flag is set to a value of binary 1 when the operation was cancelled prior to completion (e.g., system power loss, user initiated IPL). A value of binary 0 is always returned for independent ASPs.

Number of allocated auxiliary storage units in ASP is the number of configured units logically addressable by the system as units for this ASP. This is the number of configured, non-mirrored units plus the number of mirrored pairs allocated in the ASPs. The total number of units (actuator arms) on the system is the sum of the allocated auxiliary storage units plus the number of unallocated auxiliary storage units plus the number of pairs of mirrored units. For example, each 9335 enclosure represents two units. Information on these units is materialized as part of the unit information. Any two units of the same size may be associated to form a mirrored pair. Association of two units as a mirrored pair reduces the amount of logically available storage by the number of bytes contained on one of the mirrored units in the mirrored pair. This field is the number of entries which are materialized in the *unit information* section for this ASP. If an independent ASP is varied-off, it is possible that the system cannot retrieve information about the disk units in the independent ASP . Thus, a varied-off independent ASP could have 0 in this field.

Remote mirror performance mode specifies the mode in which remote mirroring operates. A value of hex 01 indicates synchronous mode. In synchronous mode, the client waits for the operation to complete

on both the source and on the target. A value of hex 02 indicates aynchronous mode. In asynchronous mode, the client waits for the operation to complete on the source and for the operation to be received on the target.

Remote mirror copy data state specifies the condition of the data on the target. A value of hex 00 indicates that remote independent ASP mirroring is not configured. A value of hex 01 indicates that the remote copy is absolutely in sync with the production copy. A value of hex 02 indicates that the remote copy contains usable data. A detached mirror copy always has usable data state. A value of hex 03 indicates that there is incoherent data state in the mirror copy and the data cannot be used.

ASP space available is the number of bytes of auxiliary storage that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation in the ASP. Note that a mirrored pair counts for only one unit. Note that a varied-off independent ASP could have 0 in this field because the system cannot determine what disk units exist in a varied-off independent ASP .

ASP event threshold specifies the minimum value for the number of bytes of auxiliary storage available in the ASP prior to the exceeded condition occurs when the ASP space available value becomes equal to or less than the ASP event threshold value. Refer to the definition of the suppress threshold exceeded event flag for more information.

The *ASP event threshold* value is calculated from the ASP event threshold percentage value by multiplying the ASP media capacity value by the ASP event threshold percentage and subtracting the product from that same capacity value.

ASP event threshold percentage specifies the auxiliary storage space utilization threshold as a percentage of the ASP media capacity. This value is used, as described above, to calculate the ASP event threshold value. This value can be modified through use of Dedicated Service Tool DASD configuration options.

Terminate immediately when out of storage indicates whether the system will be terminated immediately when a request for space occurs in the system ASP that cannot be satisfied because the system is out of storage. A value of binary 1 indicates that when a request for space in the system ASP cannot be satisfied, then the system will be terminated immediately. This field currently has meaning only in the system ASP (ASP 1). This value will always be returned as binary 0 for a user ASP.

Note:

For a physical machine with firmware level hex 00, when a request for space in the system ASP cannot be satisfied in the primary partition and the value for *terminate immediately when out of storage* is binary 1 in the primary partition, all partitions in the physical machine will terminate. When a request for space in the system ASP cannot be satisfied in a secondary partition and the value for *terminate immediately when out of storage* is binary 1 in that partition, *only* the partition in which the condition occurred will terminate. MATMATR option hex 01E0 can be used to materialize the firmware level.

For a physical machine with firmware level hex 10, *only* the partition in which the condition occurred will terminate.

A value of binary 0 indicates that when a request for space in the system ASP cannot be satisfied, then the system will not be terminated immediately, but will be allowed to continue to run however it can.

ASP contains compressed and non-compressed units flag specifies whether or not the ASP has compressed and non-compressed configured units. A value of binary 1 indicates that both compressed and non-compressed units exist in this ASP. A value of binary 0 indicates that a mix of compressed and non-compressed units does not exist in this ASP. A value of binary 0 is returned if the independent ASP is varied-off and the characteristics of the disk units in the ASP cannot be determined.

Recover overflowed basic ASP during normal mode IPL flag specifies whether or not the machine will attempt to recover the overflowed ASP data during normal mode IPLs. Overflowed data is data from the basic ASP which exists in the system ASP because there was insufficient auxiliary storage in the basic ASP. A value of binary 1 indicates that the machine will attempt to automatically recover any overflowed data for that basic ASP during normal mode IPLs. A value of binary 0 indicates that the machine will not attempt to recover the overflowed data. A value of 0 is always returned for the system ASP (ASP 1). A value of 0 is always returned for an independent ASP (since an independent ASP can never overflow its data into the system ASP).

Independent ASP specifies whether or not the ASP is an independent ASP; that is, it can be varied on and off. A value of binary 1 indicates the ASP is an independent ASP. A value of binary 0 indicates that this ASP is a basic ASP or the system ASP and cannot be varied on or off.

ASP is online flag specifies if the ASP is available to the system. If this ASP is an independent ASP, a value of binary 1 indicates the independent ASP is varied on. If this ASP is an independent ASP, a value of binary 0 indicates the independent ASP is varied off. A value of binary 1 is returned if the ASP is a basic ASP. A value of binary 1 is returned if the ASP is the system ASP.

Independent ASP address threshold exceeded flag is only valid for an Independent ASP and specifies whether or not the independent ASP address threshold, selected by the machine, has been exceeded. A value of binary 1 indicates the threshold has been exceeded and the Independent ASP is running low on addresses. A value of binary 0 indicates that the address threshold has not been exceeded.

Independent ASP is remote mirrored indicates that the independent ASP is remote mirrored. Remote independent ASP mirroring provides high availability by supporting multiple physical independent ASP copies at different sites that contain the same user data with the same virtual addresses. A value binary 0 indicates that the independent ASP is not remote mirrored. A value of binary 1 indicates that the independent ASP is remote mirrored.

ASP compression recovery policy indicates how Storage Management handles a failure condition due to a compressed disk unit being temporarily full as auxiliary storage space is reserved on the unit.

A value of binary 00 indicates that if the I/O processor can make storage space available by rearranging and further compressing data on the unit, Storage Management waits for space to be made available. When the I/O processor makes sufficient space on the compressed unit to contain the Storage Management request, the request completes successfully and the system resumes normal processing. If space can not be made available on the unit, auxiliary storage overflows from the basic ASP to the system ASP.

A value of binary 01 indicates that auxiliary storage overflows from the user ASP to the system ASP. Storage Management does not wait for the I/O processor to make storage space available on the unit.

A value of binary 10 indicates that Storage Management waits indefinitely for storage space to be made available on the unit, even if the I/O processor can not make space available on the unit. No auxiliary storage overflows from the user ASP to the system ASP.

A value of binary 00 is always returned for the system ASP (ASP 1). A value of binary 10 is always returned for independent ASPs (that is, for ASPs which can be varied on or off). An independent ASP can never have a value of binary 01 (overflow immediately) because independent ASPs are not allowed to overflow into the system ASP.

Primary ASP flag indicates that the independent ASP is a primary ASP in an ASP group. A primary ASP defines a collection of directories and contexts and may have secondary ASPs associated with it. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a primary ASP. A value of binary 0 indicates the independent ASP is not a primary ASP.

Secondary ASP flag indicates that the independent ASP is a secondary ASP in an ASP group. A secondary ASP is associated with a primary ASP. There can be many secondary ASPs associated with the same primary ASP. The secondary ASP defines a collection of directories and contexts. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a secondary ASP. A value of binary 0 indicates the independent ASP is not a secondary ASP.

UDFS ASP flag indicates that the independent ASP is a UDFS (User-defined File System) ASP. This type of independent ASP cannot be a member of an ASP group. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a UDFS ASP. A value of binary 0 indicates the independent ASP is not a UDFS ASP.

Remote mirror role identifies the current role of the physical independent ASP copy. A value of hex 00 indicates that remote independent ASP mirroring is not configured. A value of hex 01 indicates that the system does not own a physical independent ASP copy. A value of hex 02 indicates that the remote mirror role is unknown. A value of hex D7 indicates that the system owns the production copy. A value of hex D4 indicates that the system owns the mirror copy. A value of hex C4 indicates that the system owns a detached mirror copy.

Remote mirror copy state identifies the mirror state of the mirror copy. A value of hex 00 indicates that remote independent ASP mirroring is not configured. A value of hex 01 indicates that the system attempts to perform independent ASP remote mirroring when it is online. A value of hex 02 indicates that the remote independent ASP role is resuming, but the independent ASP is offline so it is not performing synchronization. A value of hex 03 indicates that the system is resuming and the independent ASP is online, so it is performing synchronization. A value of hex 04 indicates that the remote independent ASP role is detached and remote mirroring is not being performed.

ASP system storage specifies the amount of system storage currently allocated in the ASP in bytes.

ASP overflow storage indicates the number of bytes of auxiliary storage that have overflowed from a basic ASP into the system ASP. This value is valid only if the *ASP overflow storage available* field is set to a value of binary 1.

Space allocated to the error log is the number of pages of auxiliary storage that are allocated to the error log. This field only applies to the system ASP.

Space allocated to the machine log is the number of pages of auxiliary storage that are allocated to the machine log. This field only applies to the system ASP.

Space allocated to the machine trace is the number of pages of auxiliary storage that are allocated to the machine trace. This field only applies to the system ASP.

Space allocated for main store dump is the number of pages of auxiliary storage that are allocated to the main store dump space. The contents of main store are written to this location for some system terminations. This field only applies to the system ASP.

Space allocated to the microcode is the number of pages of auxiliary storage that are allocated for microcode and space used by the microcode. The space allocated to the error log, machine log, machine trace, and main store dump space is not included in this field. This field only applies to the system ASP, basic ASP, and online independent ASPs. A value of 0 is returned for offline independent ASPs.

Remote mirror synchronization priority indicates the priority assigned to synchronization between the physical copy and the mirrored copy related to the work on the system. A value of hex 00 indicates that Remote independent ASP mirroring is not configured on this independent ASP. A value of hex 10 indicates that the synchronization is given high priority, and is completed quickly at the expense of significant degradation to work on the system. A value of hex 20 indicates that the synchronization is given medium priority, and is completed at a moderate rate with some degradation to work on the

system. A value of hex 30 indicates that the synchronization is given low priority, and is completed at a slow rate with minimum degradation to work on the system.

Remote mirror encryption mode indicates the encryption mode for the remote mirrored independent ASP. A value of hex 00 indicates that Remote independent ASP mirroring is not configured on this independent ASP. A value of hex 01 indicates that the user has chosen not to encrypt the data being sent to the remote mirror site. A value of hex 02 indicates that the user has chosen to encrypt the data being sent to the remote mirror site.

Remote mirror error recovery policy indicates the error recovery policy selected by the user. A value of hex 00 indicates that remote independent ASP mirroring is not configured on this system. A value of hex 02 indicates that remote mirroring is suspended when an IASP error is detected. After suspend, if the target node becomes accessible, the system automatically resumes remote independent ASP mirroring. A value of hex 03 indicates that remote mirroring is ended when an IASP error is detected.

Remote mirror minutes until timeout is the number of minutes the system waits for a write acknowledgement from the remote system before the error recovery policy selected by the user is implemented.

Available storage lower limit is the number of bytes of available auxiliary storage in the system ASP prior to the *available storage lower limit reached* condition occurring. When the amount of auxiliary storage available in the system ASP becomes less than this amount, the *available storage lower limit reached* (hex 000C,02,08) event is signaled if it is not suppressed. Redundant signaling of this event is suppressed as indicated by the setting of the *suppress available storage lower limit reached* event flag.

Protected space capacity specifies the total number of bytes of auxiliary storage that is protected by mirroring or device parity in the ASP. Note that a varied-off independent ASP could have 0 in this field because the system could not determine what disk units exist in a varied-off independent ASP .

Unprotected space capacity specifies the total number of bytes of auxiliary storage that is not protected by mirroring or device parity in the ASP. Note that a varied-off independent ASP could have 0 in this field because the system could not determine what disk units exist in a varied-off independent ASP .

Protected space available specifies the number of bytes of protected auxiliary storage that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation in the ASP. Note that a varied-off independent ASP could have 0 in this field because the system could not determine what disk units exist in a varied-off independent ASP .

Unprotected space available specifies the number of bytes of unprotected auxiliary storage that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation in the ASP. Note that a varied-off independent ASP could have 0 in this field because the system could not determine what disk units exist in a varied-off independent ASP .

Number of addresses remaining in independent ASP contains the number of virtual addresses remaining for use by the independent ASP. This field only has meaning for an independent ASP. The information in this field is only valid if the *independent ASP address threshold exceeded* flag is set to binary 1.

Unit information is materialized in the following order:

Group 1: Configured units consisting of non-mirrored units and the first subunit of a pair of mirrored units.

Group 2: Non-configured units.

Group 3: Configured units consisting of the mates of mirrored units listed in group 1 (above).

The **unit information** is located by the **unit information offset** field which specifies the offset from the beginning of the operand 1 template to the start of the unit information. The number of entries for each of the three groups listed above is defined as follows:

Group 1: Number of non-mirrored, configured units + number of mirrored pairs

Group 2: Number of non-configured storage units (also called unallocated units).

Group 3: Number of mirrored pairs

For unallocated units the following fields contain meaningful information: device type, device identification, unit identification, unit control flags, unit relationship, and unit media capacity. The remaining fields have no meaning for unallocated units because the units are not currently in use by the system. Mirrored unit entries contain either current or last known information. The last known data consists of the mirrored unit status, disk type, disk model, unit ASP number, disk serial number, and unit address. Last known information is provided when the *mirrored pair reported* field is a binary 0.

Disk type identifies the type of disk enclosure containing this auxiliary storage unit. This is the four byte character field from the vital product data for the disk device which identifies the type of drive. For example, the value is character string '6607' for a 6607 device.

Disk model identifies the model of the type of disk enclosure containing this auxiliary storage unit. This is the four byte character field from the vital product data for the disk device which identifies the model of the drive.

Unit number uniquely identifies each non-mirrored unit or mirrored pair among the configured units. Both mirrored units of a mirrored pair have the same unit number. The value of the unit number is assigned by the system when the unit is allocated to an ASP. For unallocated units, the unit number is set to binary 0.

Unit ASP number specifies the ASP to which this unit is currently allocated. A value of 0 indicates that this unit is currently unallocated. A value of 1 specifies the system ASP. A value from 2 through 255 specifies a user ASP and correlates to the *ASP number* field in the *ASP information* entries. Values 33 to 255 specify a independent ASP. Values 2 to 32 specify a basic ASP.

Unit mirrored flag indicates that this unit number represents a mirrored pair. This bit is materialized with both mirrored units of a mirrored pair.

Mirrored unit protected flag indicates the mirror status of a mirrored pair. A value of 1 indicates that both mirrored units of a mirrored pair are active. A 0 indicates that one mirrored unit of a mirrored pair is not active. Active means that both units are on line and fully synchronized (i.e. the data is identical on both mirrored units).

Mirrored pair reported flag indicates that a mirrored unit reported as present. The mirrored unit reported present during or following IMPL. Current attachment of a mirrored unit to the system **cannot** be inferred from this bit. A 0 indicates that the mirrored unit being materialized is missing. The last known information pertaining to the missing mirrored unit is materialized. A 1 indicates that the mirrored unit being materialized has reported. The information for this reported unit is current to the last time it reported status to the system.

Mirrored unit status indicates mirrored unit status.

A value of 1 indicates that this mirrored unit of a mirrored pair is active (i.e. on-line with current data).

A value of 2 indicates that this mirrored unit is being synchronized.

A value of 3 indicates that this mirrored unit is suspended.

Mirrored unit status is stored as binary data and is valid only when the *unit mirrored* flag is on.

Unit media capacity is the space, in number of bytes of auxiliary storage, on the non-mirrored unit or mirrored pair, that is, the capacity of the unit prior to any formatting or allocation of space by the system it is attached to. For a mirrored pair, this space is the number of bytes of auxiliary storage on either one of the mirrored units. The space is identical on both of the mirrored units. Caution, do not attempt to add the capacities of the two units of a mirrored pair together. Unit media capacity is also known as ″logical capacity″. For compressed drives, the logical capacity is dynamic, and changes, depending on how well the data is compressed. A typical compressed logical capacity might be twice the drive's physical capacity.

Unit storage capacity has the same value as the *unit media capacity* for configured disk units. This value is 0 for non-configured units.

Unit space available is the number of bytes of secondary storage space that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation on the unit (or the mirrored pair). For a mirrored pair, this space is the number of bytes of auxiliary storage available on either one of the mirrored units. The space is identical on both of the mirrored units. Caution, do not attempt to add the capacities of the two units of a mirrored pair together. This value is 0 for non-configured units.

Unit space reserved for system is the total number of bytes of auxiliary storage on the unit reserved for use by the machine. This storage is not available for storing objects, redundancy data, and other internal machine data. This value is 0 for non-configured units.

Unit is device parity protected - a value of 1 indicates that this unit is device parity protected.

Subsystem is active indicates whether the array subsystem is active.

If the **unit in subsystem has failed** field is binary 1, the unit in an array subsystem being addressed has failed. Data protection for this subsystem is no longer in effect.

If the **other unit in subsystem has failed** field is binary 1, the unit being addressed is operational, but another unit in the array subsystem has failed. Data protection for this subsystem is no longer in effect.

If the **subsystem runs in degraded mode** field is binary 1, the array subsystem is operational and data protection for this subsystem is in effect, but a failure that may affect performance has occurred. It must be fixed.

If the **hardware failure** field is binary 1, the array subsystem is operational and data protection for this subsystem is in effect, but hardware failure has occurred. It must be fixed.

If the **device parity protection is being rebuilt** field is 1, the device parity protection for this device is being rebuilt following a repair action.

If the **unit is not ready** field is 1, the unit being addressed is not ready for I/O operation.

If the **unit is write protected** field is binary 1, the write operation is not allowed on the unit being addressed.

If the **unit is busy** field is binary 1, the unit being addressed is busy.

If the **unit is not operational** field is binary 1, the unit being addressed is not operational. The status of the device is not known.

If the **unit is not recognizable** field is binary 1, the unit being addressed has an unexpected status. I.e. the unit is operational, but its status returned to Storage Management from the IOP is not one of those previously described.

If the **status is not available** field is binary 1, the machine is not able to communicate with I/O processor. The status of the device is not known.

If the **unit is Read/Write protected** is binary 1, a DASD array may be in the read/write protected state when there is a problem, such as a cache problem, configuration problem, or some other array problems that could create a data integrity exposure.

If the **unit is compressed** field is binary 1, the logical capacity of the unit may be greater than its physical capacity in bytes, depending on how well the data can be compressed.

If the **do not allocate additional storage on this disk unit** field is binary 1, then new allocations will be directed away from this unit.

If the **unit is in availability parity set** field is binary 1, the unit being addressed is in a parity set optimized for availability.

If the **unit is multipath unit** field is binary 1, the unit being addressed has multipath connections to the disk unit.

Serial number specifies the serial number of the device containing this auxiliary storage unit. This is the ten character serial number field from the vital product data for the disk device.

Resource name is the unique ten-character name assigned by the system

Unit usage information specifies statistics relating to usage of the unit. For unallocated units, these fields are meaningless.

Blocks transferred to/from main storage fields specify the number of 512-byte blocks transferred for the unit since the last IMPL. These values wrap around to zero and continue counting in the case of an overflow of the field with no indication of the overflow having occurred.

Requests for data transfer to/from main storage fields specify the number of data transfer (I/O) requests processed for the unit since the last IMPL. These values wrap around to zero and continue counting in the case of an overflow of the field with no indication of the overflow having occurred. These values are not directly related to the number of blocks transferred for the unit because the number of blocks to be transferred for a given transfer request can vary greatly.

Permanent blocks transferred from main storage specifies the number of 512-byte blocks of permanent data transferred from main storage to auxiliary storage for the unit since the last IMPL. In the case of an overflow of the field, this value wraps around back to zero and continues counting, with no indication of the overflow condition having occurred.

Requests for permanent data transfer from main storage specifies the number of transfer (I/O) requests for transfers of permanent data from main storage to auxiliary storage that have been processed for the unit since the last IMPL. In the case of an overflow of the field, this value wraps around back to zero and continues counting, with no indication of the overflow condition having occurred. This value is not directly related to the permanent blocks transferred from main storage value for the unit ASP because the number of blocks to be transferred for any particular transfer request can vary greatly.

Sample count specifies the number of times the disk queue was checked to determine whether or not the queue is empty.

Not busy count specifies the number of times the disk queue was empty during the same time period that the sample count was taken.

Note that on overflow, the machine resets the following BIN(4) fields from 2,147,483,647 back to 0 without any indication of error: *blocks transferred to main storage, blocks transferred from main storage, requests for data transfer to main storage, requests for data transfer from main storage, permanent blocks transferred from main storage, requests for permanent data transfer from main storage, sample count,* and *not busy count*.

Auxiliary Storage Pool Information including offline Independent ASPs (Hex 22):

The *auxiliary storage pool information* describes the ASPs (auxiliary storage pools) which are configured within the machine. This option returns information for all ASPs including independent ASPs that are varied off. Option "Auxiliary Storage Pool Information (Short format) (Hex 1F)" (page 881) returns the same information but does not return information for independent ASPs that are varied off.

Also note that through appropriate setting of the *number of bytes provided* field for operand 1, the amount of information to be materialized for this option can be reduced thus avoiding the processing for unneeded information. For example, by setting this field to the value 48, you can get just the information for the system ASP returned.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Control information (occurs just once)	Char(16)	
16	10		Number of ASPs	UBin(2)
18	12		Reserved (binary 0)	Char(14)
32	20	ASP information (Repeated once for each ASP. Located immediately after the control information above. ASP 1, always configured, is first. Configured ASPs follow in ascending numerical order.)	[*] Char(32)	
32	20		ASP number	Char(2)
34	22		Number of allocated auxiliary storage units in ASP Note: Number of configured, non-mirrored disk units + number of mirrored pairs of disk units	UBin(2)
36	24		ASP resource name	Char(10)
46	2E		ASP control flags	Char(2)
46	2E		ASP overflow	Bit 0
46	2E		Independent ASP	Bit 1
46	2E		ASP protected	Bit 2
46	2E		User ASP MI state	Bit 3

Offset Dec	Hex	Field Name	Data Type and Length		
46	2E		Independent ASP address threshold exceeded		Bit 4
46	2E		Reserved (binary 0)		Bits 5-15
48	30		Number of addresses remaining in independent ASP	Char(8)	
56	38		ASP number of the primary ASP	Char(2)	
58	3A		Independent ASP type	Char(1)	
58	3A		Primary ASP		Bit 0
58	3A		Secondary ASP		Bit 1
58	3A		UDFS ASP		Bit 2
58	3A		Reserved (binary 0)		Bits 3-7
59	3B		Reserved (binary 0)	Char(5)	
*	*	—- End —-			

Number of ASPs is the number of ASPs configured within the machine. One, the minimum value, indicates just the system ASP exists and that there are no user ASPs configured. Up to 255 user ASPs can be configured. The system ASP always exists. This number of ASPs include the system ASP, basic ASPs and independent ASPs.

ASP information is repeated once for each ASP configured within the machine. The number of ASPs configured is specified by the *number of ASPs* field. ASP 1, the system ASP, is materialized first. Because the system ASP always exists, its materialization is always available. The information about the user ASPs is materialized after the system ASP in ascending numerical order. There may be gaps in the numerical order. For example, if user ASPs 3 and 75 are configured, the materialize will produce information on ASP 1, ASP 3, and ASP 75 in that order.

ASP number uniquely identifies the auxiliary storage pool. The ASP number may have a value from 1 through 255. A value of 1 indicates the system ASP. A value of 2 through 255 indicates a user ASP. Note that independent ASPs have a value of 33 through 255.

Number of allocated auxiliary storage units in ASP is the number of configured units logically addressable by the system as units for this ASP. This is the number of configured, non-mirrored units plus the number of mirrored pairs allocated in the ASPs. Any two units of the same capacity may be associated to form a mirrored pair. Association of two units as a mirrored pair reduces the amount of logically available storage by the number of bytes contained on one of the mirrored units in the mirrored pair.

ASP resource name specifies the name which the user has assigned to this auxiliary storage pool. Blanks (hex value 40) are returned for ASPs which do not have names. Only independent ASPs have names. The ASP name is the *resource name* in the LUD.

ASP overflow flag indicates whether or not object allocations directed into the basic ASP have overflowed into the system ASP. A value of binary 1 indicates overflow; binary 0 indicates no overflow. This flag does not apply to the system ASP and a value of binary 0 is always returned for it. This flag does not apply to independent ASPs and a value of binary 0 is always returned for independent ASPs.

Independent ASP specifies whether or not the ASP is an independent ASP; that is, a user ASP than can be varied on or off. A value of binary 1 indicates the ASP is an independent ASP. A value of binary 0 indicates that this ASP is a basic ASP (a user ASP that cannot be varied on or off).

ASP protected specifies whether or not the ASP is configured to be protected from a single disk failure. A value of binary 1 indicates that the ASP is protected. All of the disk units in this ASP must be either device parity protected or mirror protected. A value of binary 0 indicates that the disk units in the ASP are not mirror protected, and there is no requirement that the disk units in the ASP be device parity protected.

User ASP MI state indicates the state of the user ASP. A value of binary 1 indicates that the user ASP is in the 'new' state. This means that a context may be allocated in this user ASP. A value of binary 0 indicates that the user ASP is in the 'old' state. This means that there are no contexts allocated in this user ASP. This flag has no meaning for the system ASP and a value of binary 0 will always be returned for the system ASP. A value of binary 1 is always returned for independent ASPs.

Independent ASP address threshold exceeded flag is only valid for an Independent ASP and specifies whether or not the independent ASP address threshold, selected by the machine, has been exceeded. A value of binary 1 indicates the threshold has been exceeded and the Independent ASP is running low on addresses. A value of binary 0 indicates that the address threshold has not been exceeded.

Number of addresses remaining in independent ASP contains the number of virtual addresses remaining for use by the independent ASP. This field only has meaning for an independent ASP. The information in this field is only valid if the *independent ASP address threshold exceeded* flag is set to binary 1.

ASP number of the primary ASP contains the ASP number of the primary ASP. This value only has meaning for an independent ASP. If the ASP is a secondary ASP, this field contains the ASP number of the primary ASP. If the ASP is a primary ASP, this value is the same as the *ASP number*. If the ASP is a UDFS ASP or is not an independent ASP, a value of hex 0000 is returned.

Primary ASP flag indicates that the independent ASP is a primary ASP in an ASP group. A primary ASP defines a collection of directories and contexts and may have secondary ASPs associated with it. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a primary ASP. A value of binary 0 indicates the independent ASP is not a primary ASP.

Secondary ASP flag indicates that the independent ASP is a secondary ASP in an ASP group. A secondary ASP is associated with a primary ASP. There can be many secondary ASPs associated with the same primary ASP. The secondary ASP defines a collection of directories and contexts. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a secondary ASP. A value of binary 0 indicates the independent ASP is not a secondary ASP.

UDFS ASP flag indicates that the independent ASP is a UDFS (User-defined File System) ASP. This type of independent ASP cannot be a member of an ASP group. This flag only has meaning for an independent ASP. A value of binary 1 indicates the independent ASP is a UDFS ASP. A value of binary 0 indicates the independent ASP is not a UDFS ASP.

Auxiliary Storage Pool Group Information (Hex 23):

The *Auxiliary Storage Pool Group information* returns information about independent ASPs on the system. If the independent ASP can be in an Auxiliary Storage Pool Group, this option also returns primary and secondary ASP information for the ASP group which is configured within the machine.

Note that through appropriate setting of the number of bytes provided field for operand 1, the amount of information to be materialized for this option can be reduced. For example, by setting this field to provide enough bytes for the common 16 byte header plus all the fields from *ASP name* through *number of secondary ASPs*, you can get just that information and avoid the overhead of gathering the information from all of the 222 secondary ASPs which could possibly exist. You must specify at least 50 bytes in the the *number of bytes provided* field for operand 1.

Offset Dec	Hex	Field Name	Data Type and Length	
16	10	ASP name	Char(10)	
26	1A	Status	UBin(2)	
		0000 = Success		
		0001 = The independent ASP information for the input ASP name could not be located		
		0002 = The input independent ASP is not owned by this system		
		0003 = The primary and secondary ASP group information is not consistent.		
28	1C	Independent ASP type flags	Char(1)	
28	1C		Input ASP is a primary ASP	Bit 0
28	1C		Input ASP is a secondary ASP	Bit 1
28	1C		Input ASP is a UDFS ASP	Bit 2
28	1C		Reserved (binary 0)	Bits 3-7
29	1D	Reserved	Char(3)	
32	20	ASP number of primary ASP	Char(2)	
34	22	ASP name of primary ASP	Char(10)	
44	2C	Reserved	Char(4)	
48	30	Number of secondary ASPs	UBin(2)	
50	32	Reserved	Char(14)	
64	40	Secondary ASP information	[*] Char(32)	
		(Repeated once for each secondary ASP.)		
64	40		ASP number of secondary ASP	Char(2)
66	42		ASP name of secondary ASP	Char(10)
76	4C		Reserved	Char(20)
*	*	—- End —-		

ASP name is an input value that uniquely identifies the auxiliary storage pool from which the ASP group information is desired. The *ASP name* is the resource name in the LUD.

Status

•

- Hex 0000 indicates the information was successfully retrieved. Examine the *independent ASP type flags* to determine what fields in the template contain information.
- Hex 0001 indicates that the input *ASP name* could not be matched with a valid independent ASP. No other information is returned.
- Hex 0002 indicates that the input independent ASP is not owned by the system. The system cannot locate the independent ASP. The independent ASP may be switched to another node in the cluster, or the tower which contains the disk units of the independent ASP may be powered off. No other information is returned.
- Hex 0003 indicates that the input independent ASP is part of a group, but the primary and secondary ASP information is not consistent. This condition could be caused when a failure occurs during the deletion of a secondary ASP, or when a failure occurs during the creation of a secondary ASP. No other information is returned.

Input ASP is a primary ASP flag indicates whether or not the independent ASP is a primary ASP in an ASP group. A primary ASP defines a collection of directories and contexts and may have secondary ASPs associated with it. A value of binary 1 indicates the independent ASP is a primary ASP. A value of binary

0 indicates the independent ASP is not a primary ASP. If the *input ASP is a primary ASP*, the following fields are also returned: *ASP number of primary ASP,ASP name of primary ASP,number of secondary ASPs,* and *secondary ASP information*.

Input ASP is a secondary ASP flag indicates whether or not the independent ASP is a secondary ASP in an ASP group. A secondary ASP is associated with a primary ASP. There can be many secondary ASPs associated with the same primary ASP. The secondary ASP defines a collection of directories and contexts. A value of binary 1 indicates the independent ASP is a secondary ASP. A value of binary 0 indicates the independent ASP is not a secondary ASP. If the *input ASP is a secondary ASP*, the following fields are also returned: *ASP number of primary ASP,ASP name of primary ASP,number of secondary ASPs,* and *secondary ASP information*.

Input ASP is a UDFS ASP flag indicates whether or not the independent ASP is a UDFS (User-defined File System) ASP. This type of independent ASP cannot be a member of an ASP group. A value of binary 1 indicates the independent ASP is a UDFS ASP. A value of binary 0 indicates the independent ASP is not a UDFS ASP. If the *input ASP is a UDFS ASP*, no other data is returned.

ASP name of primary ASP specifies the name which the user has assigned to the primary auxiliary storage pool. The ASP name is the resource name in the LUD.

ASP number of primary ASP contains the ASP number of the primary ASP. This field has a value from 33 through 255.

Number of secondary ASPs is the number of secondary ASPs associated with the primary ASP. There are this many *secondary ASP information* entries (below).

Secondary ASP information is repeated once for each secondary ASP associated with the primary ASP. The number of entries is specified by the *number of secondary ASPs* field. The *secondary ASP Information* entries are not sorted in any particular order.

ASP number of secondary ASP uniquely identifies the auxiliary storage pool. The ASP number has a value from 33 through 255.

ASP name of secondary ASP specifies the name which the user has assigned to the secondary auxiliary storage pool. The ASP name is the resource name in the LUD.

Dynamic Thread Resources Affinity Adjustment (Hex 24):

All threads in the machine have affinity with a machine-determined portion of its resources, including processors and main memory. The portion is determined when the thread is initiated. **Dynamic thread resources affinity adjustment** controls whether the machine may make adjustments to these portions at a later time. Hex 00 is the default value.

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Dynamic thread resources affinity adjustment	Char(1)
		Hex 00 = Enable. The machine may make adjustments to the portion of its resources with which individual threads have affinity.	
		Hex 01 = Disable.The machine will not make adjustments to the portion of its resources with which any thread has affinity.	
17	11	Reserved (binary 0)	Char(3)

Offset Dec	Hex	Field Name	Data Type and Length
20	14	—- End —-	

Auxiliary Storage Pool Space Information (Hex 25):

The *auxiliary storage space information* describes the ASPs (auxiliary storage pools) which are configured within the machine. This option does not return information for independent ASPs which are varied off.

This materialize provides an alternative to Hex 12 for those users who want only the capacity and available space returned.

Modification of most of the auxiliary storage configuration is performed using functions available in the Dedicated Service Tool (DST).

Offset Dec	Hex	Field Name	Data Type and Length	
16	10	Control information	Char(64)	
		(occurs just once)		
16	10		Number of ASPs	UBin(2)
18	12		Reserved (binary 0)	Char(62)
80	50	ASP information	[*] Char(64)	
		(Repeated once for each ASP. Located immediately after the control information above. ASP 1, always configured, is first. Configured user ASPs follow in ascending numerical order.)		
80	50		ASP number	Char(2)
82	52		Reserved (binary 0)	Char(6)
88	58		ASP space capacity (Bound program)	UBin(8)
88	58		ASP space capacity (Non-Bound program)	Char(8)
96	60		ASP space available (Bound program)	UBin(8)
96	60		ASP space available (Non-Bound program)	Char(8)
104	68		Reserved (binary 0)	Char(40)
*	*	—- End —-		

Number of ASPs is the number of ASPs configured within the machine. One, the minimum value, indicates just the system ASP exists and that there are no user ASPs configured. Up to 254 user ASPs can be configured. The system ASP always exists. The *number of ASPs* includes the system ASP, user ASPs which are basic ASPs (that is, user ASPs which cannot be varied on or off), and independent ASPs which are currently varied on to this system.

ASP information is repeated once for each configured ASP within the machine that is online. The number of ASPs configured is specified by the *number of ASPs* field. ASP 1, the system ASP, is materialized first. Because the system ASP always exists, its materialization is always available. The user ASPs which are configured are materialized after the system ASP in ascending numerical order. There may be gaps in the numerical order. That is, if just user ASPs 3 and 5 are configured, only information for them is materialized producing information on just ASP 1, ASP 3 and ASP 5 in that order.

ASP number uniquely identifies the auxiliary storage pool. The ASP number may have a value from 1 through 255. A value of 1 indicates the system ASP. A value of 2 through 255 indicates a user ASP. Note that independent ASPs have a value of 33 through 255.

ASP space capacity specifies the total space, in number of bytes of auxiliary storage, on the storage media allocated to the ASP. This is just the sum of the unit media capacity fields for (1) the units allocated to the ASP or (2) the mirrored pairs in the ASP. Note that a mirrored pair counts for only one unit.

ASP space available is the number of bytes of auxiliary storage that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation in the ASP. Note that a mirrored pair counts for only one unit.

Processor Utilization Data (Hex 26):

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Processor utilized time since IPL (Bound program)	UBin(8)	
16	10		Processor utilized time since IPL (Non-Bound program)	Char(8)
24	18	Processor configured available processing time since IPL (Bound program)	UBin(8)	
24	18		Processor configured available processing time since IPL (Non-Bound program)	Char(8)
32	20	Processor uncapped available time since IPL (Bound program)	UBin(8)	
32	20		Processor uncapped available time since IPL (Non-Bound program)	Char(8)
40	28	Secondary workload utilized time since IPL (Bound program)	UBin(8)	
40	28		Secondary workload utilized time since IPL (Non-Bound program)	Char(8)
48	30	Database utilized time since IPL (Bound program)	UBin(8)	
48	30		Database utilized time since IPL (Non-Bound program)	Char(8)
56	38	Database threshold	UBin(2)	
58	3A	Database limit	UBin(2)	
60	3C	Reserved (binary 0)	Char(4)	
64	40	Interactive utilized time since IPL (Bound program)	UBin(8)	
64	40		Interactive utilized time since IPL (Non-Bound program)	Char(8)
72	48	Interactive available time since IPL (Bound program)	UBin(8)	
72	48		Interactive available time since IPL (Non-Bound program)	Char(8)
80	50	Interactive threshold	UBin(2)	
82	52	Interactive limit	UBin(2)	
84	54	Current processing capacity	UBin(4)	
88	58	Current number of processors	UBin(2)	
90	5A	Reserved (binary 0)	Char(6)	
96	60	—- End —-		

Processor utilized time since IPL is the sum of all time, in milliseconds, utilized by all processors since IPL.

Processor configured available processing time since IPL is the total amount of configured processor time, in milliseconds, available since IPL. In a dedicated partition, the configured available processing time is the elapsed time times the number of processors, tracked over time as the configuration changes. In a shared partition, the configured available processing time is the elapsed time times the number of shared processor units, tracked over time as the configuration changes.

Processor uncapped available time since IPL is the total amount of processing time, in milliseconds, available since IPL, tracked over time as the configuration changes. The total available time includes configured available time and an additional amount of shared processor pool available time that is available to the partition because it is uncapped. The uncapped available time represents the upper limit on the partition's potential utilized time based on the configuration of the partition and the shared pool. In dedicated and capped shared processor partitions, the processor uncapped available time since IPL equals the processor configured available processing time since IPL. In an uncapped shared processor partition, the uncapped available time is the elapsed time multiplied by the minimum of the number of processors in the partition and the number of processors in the shared pool, tracked over time as the configuration changes.

Secondary workload utilized time since IPL is the total processor time, in milliseconds, used for workloads that can not fully exploit dedicated server resources, since IPL. If a system is not a dedicated server, a value of hex 0000000000000000 is returned.

Database utilized time since IPL is the total processor time, in milliseconds, used performing database processing, since IPL. If the system does not support this metric, a value of hex 0000000000000000 is returned. If the system does support this and needs to return a value of 0, a value of hex 0000000000000001 is returned.

Database threshold is the highest level of database processor utilization which can be sustained without causing a disproportionate increase in system overhead. The value returned is the fraction of processor capacity, expressed in tenths of a percent. For example, a value of 237 means that the threshold is 23.7%. On a machine with no limit on database utilization, the value returned will be 1000 (100%).

Database limit is the maximum sustainable level of database processor utilization. The value returned is the fraction of processor capacity, expressed in tenths of a percent. For example, a value of 275 means that the limit is 27.5%. On a machine with no limit on database utilization, the value returned will be 1000 (100%).

Interactive utilized time since IPL is the total processor time, in milliseconds, used by interactive processes, since IPL. If the system does not support this metric, a value of hex 0000000000000000 is returned. If the system does support this and needs to return a value of 0, a value of hex 0000000000000001 is returned.

Interactive available time since IPL is the total processor time available, in milliseconds, to interactive processes, since IPL. If the system does not support this metric, a value of hex 0000000000000000 is returned. If the system does support this and needs to return a value of 0, a value of hex 0000000000000001 is returned.

Interactive threshold is the highest level of interactive processor utilization which can be sustained without causing a disproportionate increase in system overhead. The value returned is the fraction of processor capacity, expressed in hundredths of a percent. For example, a value of 2379 means that the threshold is 23.79%. On a machine with no limit on interactive utilization, the value returned will be 10000 (100%).

Interactive limit is the maximum sustainable level of interactive processor utilization. The machine determines the interactive limit based on the interactive feature. The value returned is the fraction of processor capacity, expressed in hundredths of a percent. For example, a value of 4572 means that the limit is 45.72%. On a machine with no limit on interactive utilization, the value returned will be 10000 (100%).

Current processing capacity specifies the current processing capacity of the partition. The value returned for this attribute is accurate to a hundredth of a physical processor. For example, a value of 233 means that the partition's current processing capacity is equivalent to 2.33 physical processors. See MATMATR option hex 01E0 for a complete description.

Current number of processors is the number of virtual processors that are currently enabled to run for this partition.

Note: The number of virtual processors allocated to the current partition that are active for the current IPL can also be materialized using option hex 13 on MATRMD and option hex 01E0 on MATMATR.

Shared Processor Pool Information (Hex 27):

This option returns information on the shared processor pool. Shared processor pool information is not available for all hardware models.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Shared processor pool available time since IPL (Bound program)	UBin(8)	
16	10		Shared processor pool available time since IPL (Non-Bound program)	Char(8)
24	18	Shared processor pool utilized time since IPL (Bound program)	UBin(8)	
24	18		Shared processor pool utilized time since IPL (Non-Bound program)	Char(8)
32	20	Number of processors in shared processor pool	UBin(2)	
34	22	Reserved (binary 0)	Char(2)	
36	24	Materialization status **Hex 00 =** Data returned successfully **Hex 01 =** Data not returned, data not available for hardware model **Hex 02 =** Data not returned, partition is not in a shared pool **Hex 03 =** Data not returned, partition is not allowed to get shared pool information	Char(1)	
37	25	Reserved (binary 0)	Char(3)	
40	28	—- End —-		

Shared processor pool available time since IPL is the total amount of processor time, in milliseconds, available in the shared processor pool since IPL.

Shared processor pool utilized time since IPL is the total amount of processor time, in milliseconds, utilized in the shared processor pool since IPL.

Number of processors in shared processor pool is the number of processors that are allocated to the shared pool in which the partition is executing. The number of processors in the shared pool is always less than or equal to the number of processors in the machine.

Materialization status indicates whether or not the shared processor pool information was returned.

Multiprocessor utilizations (Hex 28):

Offset Dec	Offset Hex	Field Name	Data Type and Length
16	10	Maximum number of active processors in the partition	UBin(2)
18	12	Number of active processors in the partition	UBin(2)
20	14	Number of table entries	UBin(2)
22	16	Reserved (binary 0)	Char(26)
48	30	Table entry (repeated *number of table entries* times)	[*] Char(48)
48	30		Processor utilized time since IPL (Bound program)
48	30		Processor utilized time since IPL (Non-Bound program)
56	38		Processor configured available time since IPL (Bound program)
56	38		Processor configured available time since IPL (Non-Bound
64	40		Processor uncapped available time since IPL (Bound program)
64	40		Processor uncapped available time since IPL (Non-Bound p
72	48		Processor id
74	4A		Processor status flags
74	4A		Processor installed **0 =** Processor is not installed **1 =** Processor is installed
74	4A		Processor active **0 = Processor is inactive** **1 = Processor is active**
74	4A		Reserved (binary 0)
75	4B		Reserved (binary 0)
*	*	—- End —-	

Maximum number of active processors in the partition is the maximum number of virtual processors that can be active on the current IPL of the partition.

Number of active processors in the partition is the number of virtual processors currently active in the partition. It will always be less than or equal to the *maximum number of active processors in the partition*.

Number of table entries is the count of the number of *table entries* returned. No partial table entries will be returned.

Table entry contains the amount of time, in milliseconds, the individual processor has been available for and utilized for work. It also contains *processor status flags* indicating the current status of the processor.

Processor utilized time since IPL is the sum of all time, in milliseconds, utilized by this processor since IPL.

Processor configured available processing time since IPL is the total amount of configured processor time, in milliseconds, available to this processor since IPL. In a dedicated partition, the configured available processing time is the elapsed time. In a shared partition, configured available processing time is the elapsed time times the ratio of shared processor units to processors, tracked over time as the configuration changes.

Processor uncapped available time since IPL is the total amount of processing time, in milliseconds, available since IPL, tracked over time as the configuration changes. The total available time includes configured available time and an additional amount of shared processor pool available time that is available to the partition because it is uncapped. The uncapped available time represents the upper limit on the partition's potential utilized time based on the configuration of the partition and the shared pool. In dedicated and capped shared processor partitions, the processor uncapped available time since IPL equals the processor configured available processing time since IPL. In an uncapped shared processor partition, the uncapped available time is the elapsed time multiplied by the minimum of the number of processors in the partition and the number of processors in the shared pool, tracked over time as the configuration changes.

Processor id identifies the virtual processor.

Processor status flags indicates the current status of the virtual processor.

The **processor installed** field indicates whether or not the processor is installed on the system. A value of binary 0 indicates that the virtual processor is unavailable for the duration of the IPL. A value of binary 1 indicates the processor is installed and may be varied on/off for the duration of the IPL.

The **processor active** field indicates whether or not the processor is active on the system. A value of binary 0 indicates the processor is currently varied off or is not installed on the system. A value of binary 1 indicates the processor is currently varied on.

Materialize machine resource portions (Hex 29): *Materialize machine resource portions* describes the machine-determined portions of the internal machine processing resources.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Reserved (binary 0)	Char(12)	
28	1C	Number of entries	UBin(4)	
32	20	Entry	[*] Char(32)	
		(repeated *number of entries* times)		
32	20		Machine resource portion identifier	UBin(4)
36	24		Weight	UBin(2)
38	26		Reserved (binary 0)	Char(26)
*	*	—- End —-		

Number of entries is the count of the number of *entries* returned. No partial entries will be returned. The machine always has at least one portion which may be materialized.

Machine resource portion identifier identifies a machine-determined portion of the internal machine processing resources.

Weight is an indication of the relative importance of one portion of the internal machine processing resources over another. The relative importance of two portions of the internal machine processing resources is determined by the proportions of their weights. For example if the weight of one portion is twice that of another portion, then the first portion is twice as important as the second.

Materialize interrupt polling control (Hex 2A):

Note: IOPless device drivers do not use interrupt polling.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
16	10	Interrupt polling control	Char(8)	
16	10		Pending interrupt polling control	Char(1)
			Hex 00 = The system is interrupted when an I/O operation completes. Any work being done on the processor handling the interrupt is suspended until the interrupt service routine completes.	
			Hex 01 = The system periodically reads the hardware interrupt registers to determine if any I/O has completed. This allows work that is currently being done by the system to proceed without being interrupted. It may allow the system to complete work more efficiently, but may also increase response time.	
17	11		Current interrupt polling control	Char(1)
			Hex 00 = The system is interrupted when an I/O operation completes. Any work being done on the processor handling the interrupt is suspended until the interrupt service routine completes.	
			Hex 01 = The system periodically reads the hardware interrupt registers to determine if any I/O has completed. This allows work that is currently being done by the system to proceed without being interrupted. It may allow the system to complete work more efficiently, but may also increase response time.	
			Hex 02 = The system is interrupted when an I/O operation completes. Any work being done on the processor handling the interrupt is suspended until the interrupt service routine completes. A pending request to poll interrupts was not fulfilled on the current IPL because the partition uses shared processors. In this case, the value of the *pending interupt polling control* is hex 01.	
18	12		Reserved (binary 0)	Char(6)
24	18	—- End —-		

Pending interrupt polling control reflects the value of *interrupt polling control* requested. If the value of *pending interrupt polling control* is not the same as the value of *current interrupt polling control*, the *pending interrupt polling control* value will take effect on the next IPL.

Current interrupt polling control reflects the value of *interrupt polling control* that is currently in effect.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Selected Locks (MATSELLK)

Op Code (Hex)	Operand 1	Operand 2
033E	Receiver	Object, object location or space location

Operand 1: Space pointer.

Operand 2: System pointer, object pointer or space pointer data object.

Bound program access

```
Built-in number for MATSELLK is 53.
MATSELLK (
          receiver                                  : address
          object_or_object_location_space_location  : address of system pointer OR
                                                      address of object pointer OR
                                                      address of space pointer(16)
)
```

Description: The locks for the *object, object location, or space location* referenced by operand 2 are materialized. The locks materialized are the thread scoped locks held by the current thread and the process scoped locks held by the process containing the current thread. The format of the materialization template is as follows:

Offset				
Dec	Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
4	4		Number of bytes available for materialization	Bin(4)
8	8	Cumulative lock status for all locks on operand 2	Char(1)	
8	8		Lock state	Char(1)
8	8		LSRD	Bit 0
8	8		LSRO	Bit 1
8	8		LSUP	Bit 2
8	8		LEAR	Bit 3
8	8		LENR	Bit 4
8	8		Reserved (binary 0)	Bits 5-7
9	9	Reserved	Char(3)	
12	C	Number of lock entries	Bin(2)	
14	E	Return Format	Char(1)	
14	E		Reserved (Binary 0)	Bits 0-3
14	E		Use expanded results	Bit 4
14	E		Do not return locks held by a transaction	Bit 5
14	E		Do not return locks held by a process	Bit 6
14	E		Do not return locks held or waited on by a thread	Bit 7
15	F	Reserved	Char(1)	
16	10	Lock status	[*] Char(32)	
		(repeated *number of lock entries* times) (If the *use expanded results* flag is binary 1 in the *return format* field then the size of each lock status entry is 32 bytes, otherwise the lock status entry is 2 bytes.		
16	10		Lock state	Char(1)
			Hex 80 = LSRD lock request	
			Hex 40 = LSRO lock request	
			Hex 20 = LSUP lock request	
			Hex 10 = LEAR lock request	
			Hex 08 = LENR lock request	
			All other values are reserved	
17	11		Status of lock	Char(1)
17	11		Lock held by a transaction control structure	Bit 0
17	11		Lock scope	Bit 1
			0 = Lock is held by the process containing the current thread	
			1 = Lock is held by the current thread	
17	11		Reserved (binary 0)	Bits 2-5
17	11		Implicit lock	Bit 6
			0 = Not implicit lock	
			1 = Is implicit lock	
17	11		Reserved (binary 1)	Bit 7

Offset Dec	Hex	Field Name	Data Type and Length	
			(The following fields are only returned if the *return format* field specifies *use expanded results*.)	
18	12		Reserved (binary 0)	Char(14)
32	20		Suspend pointer	Suspend Pointer
*	*	—- End —-		

The first 4 bytes of the materialization identifies the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identifies the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions are signaled in the event that the *receiver* contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception described previously.

A space pointer machine object cannot be specified for operand 2.

The maximum number of locks that can be materialized with this instruction is 32,767. No exception will be signaled if more than 32,767 exist and only the first 32,767 locks found will be materialized.

If a space pointer to a teraspace storage location is specified for operand 2, and if the teraspace storage location is mapped to single level storage, then the locks on that single level storage will be materialized, otherwise locks on the teraspace storage location will be materialized.

Warning: Temporary Level 3 Header

Authorization

•

• Execute

–

– Context referenced by address resolution

Lock Enforcement

•

• Materialize

–

– Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

28 Process/Thread State

2802 Process Control Space Not Associated with a Process

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Space Attributes (MATS)

Op Code (Hex)	Operand 1	Operand 2
0036	Receiver	Space object

Operand 1: Space pointer.

Operand 2: System pointer.

Bound program access

```
Built-in number for MATS is 27.
MATS (
      receiver       : address
      space_object   : address of system pointer
)
```

Description: The current attributes of the *space object* specified by operand 2 are materialized into the *receiver* specified by operand 1.

The template identified by operand 1 must be 16-byte aligned in the space. The format of the materialization is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Object identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
40	28	Object creation options	Char(4)	
40	28		Existence attribute	Bit 0
			0 = Temporary	
			1 = Permanent	
40	28		Space attribute	Bit 1
			0 = Fixed-length	
			1 = Variable-length	
40	28		Context	Bit 2
			0 = Addressability not in context	
			1 = Addressability in context	
40	28		Access group	Bit 3
			0 = Not member of access group	
			1 = Member of access group	
40	28		Reserved (binary 0)	Bits 4-12
40	28		Initialize space	Bit 13
			0 = Initialize	
			1 = Do not initialize	
40	28		Automatically extend space	Bit 14
			0 = No	
			1 = Yes	
40	28		Hardware storage protection level	Bits 15-16
			00 = Reference and modify allowed for user state programs	
			01 = Only reference allowed for user state programs	
			10 = Invalid (undefined)	
			11 = No reference or modify allowed for user state programs	
40	28		Reserved (binary 0)	Bits 17-20
40	28		Always enforce hardware storage protection of this space	Bit 21
			0 = Enforce hardware storage protection of this space only when hardware storage protection is enforced for all storage.	
			1 = Enforce hardware storage protection of this space at all times.	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
40	28		Reserved (binary 0)	Bits 22-31
44	2C	Reserved (binary 0)	Char(2)	
46	2E	ASP number	Char(2)	
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
53	35		Space alignment	Bit 0
			0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space.	
			1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space.	
			Ignore the value of this field when the *machine chooses space alignment* field has a value of 1.	
53	35		Reserved (binary 0)	Bit 1
53	35		Spread the space object	Bit 2
			0 = The space object may be on one storage device.	
			1 = The space object may be spread across multiple storage devices.	
53	35		Machine chooses space alignment	Bit 3
			0 = The space alignment indicated by the *space alignment* field is in effect.	
			1 = The machine chose the space alignment most beneficial to performance, which may have reduced maximum space capacity. The alignment chosen is a multiple of 512. Check the *maximum size of space* field value. Ignore the value of the *space alignment* field.	
53	35		Reserved (binary 0)	Bit 4
53	35		Main storage pool selection	Bit 5
			0 = Process default main storage pool is used for object.	
			1 = Machine default main storage pool is used for object.	
53	35		Transient storage pool selection	Bit 6

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = Default main storage pool (process default or machine default as specified for main storage pool selection) is used for object.	
			1 = Transient storage pool is used for object.	
53	35		Obsolete This field is no longer used and will be ignored.	Bit 7
53	35		Unit number	Bits 8-15 +
53	35		Reserved (binary 0)	Bits 16-23
56	38		Expanded transfer size advisory	Char(1)
57	39	Reserved (binary 0)	Char(7)	
64	40	Context	System pointer	
80	50	Access group	System pointer	
96	60	Reserved (binary 0)	Char(16)	
112	70	Maximum size of space	Bin(4)	
116	74	—- End —-		

The first 4 bytes that are materialized identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes a *materialization length invalid* (hex 3803) exception.

The second 4 bytes that are materialized identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the materialization length exception described previously) are signaled in the event that the receiver contains insufficient area for the materialization.

See the Create Space (CRTS) instruction for descriptions of most of these fields.

The **maximum size of space** field returns the maximum number of bytes which may be contained in the space. For fixed-length spaces, the current size is the maximum size. This value is the actual maximum size, not the size specified for *largest size needed for space* on CRTS.

This instruction cannot be used to materialize the *public authority specified* creation option, the *initial owner specified* creation option, the *process temporary space accounting* creation option, or the template extension which can be specified on space creation. The Materialize Authority (MATAU) instruction can be used to materialize the current public authority for the space. The Materialize System Object (MATSOBJ) instruction can be used to materialize the current owner of the space.

Warning: Temporary Level 3 Header

Authorization Required

•
- Operational or space authority
  –
  – Operand 2
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•
- Materialize
  –
  – Operand 2
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize System Object (MATSOBJ)

Op Code (Hex)	Operand 1	Operand 2
053E	Receiver	Object

Operand 1: Space pointer.

Operand 2: System pointer.

Bound program access
Built-in number for MATSOBJ is 91. MATSOBJ (receiver : address object : address of system pointer)

Description: This instruction materializes the identity and size of a system object addressed by the system pointer identified by operand 2. It can be used whenever addressability to a system object is contained in a system pointer.

The format of the materialization is:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided by the user	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Object state attributes	Char(2)	
8	8		Suspended state	Bit 0
			0 = Not suspended	
			1 = Suspended	
8	8		Damage state	Bit 1
			0 = Not damaged	
			1 = Damaged	
8	8		Partial damage state	Bit 2
			0 = No partial damage	
			1 = Partial damage	
8	8		Existence of addressing context	Bit 3
			0 = Not addressed by a temporary context	
			1 = Addressed by a temporary context	
8	8		Dump for previous release permitted	Bit 4
			0 = Dump for previous release not permitted	
			1 = Dump for previous release permitted	
8	8		Object compressed	Bit 5
			0 = Object not compressed	
			1 = Object compressed (partially or completely)	
8	8		ASP overflow	Bit 6
			0 = No part of the object has overflowed its ASP	
			1 = Some part of the object has overflowed its ASP	
8	8		Object requires format change for this machine implementation	Bit 7

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = No conversion is required	
			1 = Object must be converted before it can be used on this machine implementation	
8	8		Reserved (binary 0)	Bits 8-15
10	A	Context identification	Char(32)	
10	A		Context type	Char(1)
11	B		Control subtype	Char(1)
12	C		Context name	Char(30)
42	2A	Object identification	Char(32)	
42	2A		Object type	Char(1)
43	2B		Object subtype	Char(1)
44	2C		Object name	Char(30)
74	4A	Timestamp of creation	Char(8)	
82	52	Size of associated space	Bin(4)	
86	56	Object size	Bin(4)	
90	5A	Owning user profile identification	Char(32)	
90	5A		User profile type	Char(1)
91	5B		User profile subtype	Char(1)
92	5C		User profile name	Char(30)
122	7A	Timestamp of last modification	Char(8)	
130	82	Recovery options	Char(4)	
130	82		Machine internal use	Char(2)
132	84		ASP number	Char(2)
134	86	Performance class	Char(4)	
138	8A	Initial value of space	Char(1)	
139	8B	Object audit attribute	Char(1)	
		Hex 00 = No audit for this object		
		Hex 02 = Audit change for this object		
		Hex 03 = Audit read and change for this object		
		Hex 04 = Audit read and change for this object if the user profile is being audited		
140	8C	Sign state	Char(1)	
		Hex 00 = Object not digitally signed		
		Hex 01 = Object digitally signed		
141	8D	Signed by a system-trusted source	Char(1)	
		Hex 00 = Object not digitally signed by a system-trusted source		
		Hex 01 = Object digitally signed by a system-trusted source		
142	8E	Object authority list (AL) status	Bin(2)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
		0 = Object not in an authority list		
		1 = Object in an authority list		
144	90	Authority list identification	Char(48)	
144	90		Authority list (AL) status	Bin(2)
			0 = Valid authority list	
			1 = Damaged authority list	
			2 = Destroyed authority list (no name below)	
146	92		Reserved	Char(14)
160	A0		Authority list type	Char(1)
161	A1		Authority list subtype	Char(1)
162	A2		Authority list name	Char(30)
192	C0	Dump for previous release reason code	Char(8)	
200	C8	Maximum possible associated space size	Bin(4)	
204	CC	Timestamp of last use of object	Char(8)	
212	D4	Count of number of days object was used	UBin(2)	
214	D6	Program or module attributes	Char(2)	
214	D6		State provided	Bit 0
			0 = No program/module state value	
			1 = Program/module state value present	
214	D6		Program executable portion compression status	Bit 1
			0 = Program's executable portion is not compressed	
			1 = Program's executable portion is compressed	
214	D6		Program extended observability storage area existence	Bit 2
			0 = Program's extended observability storage area does not exist or can no longer be materialized	
			1 = Program's extended observability storage area does exist and can be materialized	
214	D6		Program extended observability storage area compression status If the compression status of the extended observability storage area of the program is uncompressed, it is not a guarantee that the extended observability storage area exists. The *extended observability storage area existence* should be checked first to ensure that it currently exists in the program.	Bit 3
			0 = Program's extended observability storage area is not compressed	
			1 = Program's extended observability storage area is compressed	
214	D6		Reserved (binary 0)	Bits 4-7
215	D7		Type of program	Char(1)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			Hex 00 = Non-bound program	
			Hex 01 = Bound program	
			Hex 02 = Bound service program	
			Hex 04 = Java program	
216	D8	Domain of object	Char(2)	
218	DA	State for program or module	Char(2)	
220	DC	MI-supplied information	Char(8)	
228	E4	Earliest compatible release	Char(2)	
228	E4		Reserved	Bits 0-3
228	E4		Version	Bits 4-7
228	E4		Release	Bits 8-11
228	E4		Modification level	Bits 12-15
230	E6	Object size in basic storage units	UBin(4)	
234	EA	Primary group identification	Char(32)	
234	EA		Profile type	Char(1)
235	EB		Profile subtype	Char(1)
236	EC		Profile name	Char(30)
266	10A	Hardware storage protection	Char(1)	
266	10A		Hardware storage protection of object's encapsulated part	Bits 0-1
266	10A		Hardware storage protection of associated space	Bits 2-3
266	10A		Creation hardware storage protection	Bits 4-5
266	10A		Hardware storage protection always enforced for object	Bit 6
			0 = Hardware storage protection of this object's encapsulated part is enforced only when hardware storage protection is enforced for all storage.	
			1 = Hardware storage protection of this object's encapsulated part is enforced at all times.	
266	10A		Hardware storage protection always enforced for associated space	Bit 7
			0 = Hardware storage protection is enforced for this object's primary associated space only when hardware storage protection is enforced for all storage.	
			1 = Hardware storage protection is enforced for this object's primary associated space at all times.	
267	10B	Reserved	Char(1)	
268	10C	File identifier	UBin(4)	
272	110	Generation identifier	UBin(4)	
276	114	Reserved	Char(12)	

Offset Dec	Hex	Field Name	Data Type and Length
288	120	Parent of attached object	System pointer
304	130	Number of signers	UBin(4)
308	134	Reserved	Char(36)
344	158	—- End —-	

Additional Description: This instruction will tolerate a damaged object referenced by operand 2 when operand 2 is a resolved pointer. The instruction will not tolerate a damaged context(s) or damaged programs when resolving pointers. Also, as a result of damage or abnormal machine termination, this instruction can indicate that an object is addressed by a context, when in fact the context will not show this as an addressed object.

The **existence of addressing context** field indicates whether the previously (or currently) addressing context was (is) temporary. This field is 0 if the object was (is) not addressed by a temporary context.

The **dump for previous release permitted** field will indicate if the object is eligible for a dump for previous request. [1] When this field indicates that the object is not eligible, the **dump for previous release reason code** can be used to determine why the object is not eligible.

The **object compressed** field indicates whether the encapsulated part of the object is either partially or completely compressed. The encapsulated part(s) of some object types can be compressed by object-specific create or modify instructions.

For program objects other than Java program objects, additional compression information is provided by the **program executable portion compression status** and the **program extended observability storage area compression status** fields. This program compression information is also available from the Materialize Program (MATPG) or the Materialize Bound Program (MATBPGM) instructions. However, the use of those instructions may cause the program object to be temporarily decompressed to obtain this compression information. By using the program compression information provided in this instruction instead of using Materialize Program (MATPG) or the Materialize Bound Program (MATBPGM) instructions, this temporary decompression of the program object can be avoided.

For Java program objects, additional compression information is not available.

For objects other than programs, use the object-specific materialization instruction to determine exactly which part(s) of the object are compressed.

The **ASP overflow** field indicates whether any part of the object is stored in an ASP other than the ASP specified at the time the object was created. If any object created in one ASP has parts that are in a different ASP (due to lack of sufficient available storage in the original ASP), then none of the objects in the first ASP are protected in the event of a failure of any other ASP in the system. By deleting objects that have overflowed, however, it may be possible to eliminate the ASP overflow condition and restore the protection that ASPs provide. Use the object-specific materialization instruction for this type of object to determine what ASP was specified at the time the object was created.

The **object requires format change for this machine implementation** field is set for program or module objects. It indicates whether the program or module is in the correct format for the current machine implementation or if they need to be converted (retranslated) before use. For all other object types, the field will have a value of 0.

If the object addressed by the system pointer specifies that it is not addressed by a context or if the context is destroyed, the *context type* field is hex 00. If the object is addressed by the machine context, a *context type* field of hex 81 is returned. No verification is made that the specified context actually addresses the object.

Valid **object type** fields and their meanings are:

Value (Hex)	Object Type
01	Access group
02	Program
03	Module
04	Context
06	Byte string space
07	Journal space
08	User profile
09	Journal port
0A	Queue
0B	Data space
0C	Data space index
0D	Cursor
0E	Index
0F	Commit block
10	Logical unit description
11	Network description
12	Controller description
13	Dump space
14	Class of service description
15	Mode description
16	Network interface description
17	Connection list
18	Queue space
19	Space
1A	Process control space
1B	Authority list
1C	Dictionary
1D	Auxiliary server
1E	Byte stream file
20	XOM object
21	Composite object group
23	Transaction control structure

The **timestamp** field is materialized as an 8-byte unsigned binary number. See "Standard Time Format" on page 1272 for additional information on the format of timestamps. The **timestamp of creation** field is implicitly set when an object is created.

If the object has an associated space, the **maximum possible associated space size** field will be returned with a value which represents the maximum size to which the associated space can be extended. This value depends on the internal packaging of the object and its associated space as well as (possibly) the maximum space size field as optionally specified during the create of the object.

The **object size** field will contain the size of the object in bytes up to a value of 2G-1 (2,147,483,647). If the object's size is greater than this, a value of zero will be returned in the *object size* field. In this case, the **object size in basic storage units** field should be used to get the object's actual size. This field will always contain the object's true size in number of basic storage units.

If the object is a temporary object and is, therefore, owned by no user profile, the *user profile type* field is assigned a value of hex 00.

The **timestamp of last modification** field is implicitly set, except for the objects restricted below, by any instruction or IMPL function that modifies or attempts to modify an object attribute value or an object state. The timestamp of last modification field is only ensured as part of the normal ensuring of objects.

Implicit setting of the timestamp of last modification field is restricted for the following objects and will only occur for generic, nonobject specific, operations on them.

•
- Logical unit description
- Controller description
- Network description
- Access group
- Queue

No modification timestamp will be provided for the following objects and a value of zero will be returned in the materialization template for the *modification timestamp*.

•
- Process control space

ASP number uniquely identifies the auxiliary storage pool. The *ASP number* may have a value from 0 through 255. A value of 0 indicates the system ASP. A value of 2 through 255 indicates a user ASP. A value of 2 through 32 indicates a basic ASP. A value of 33 through 255 indicates an independent ASP.

Sign state indicates whether or not the object is digitally signed.

Signed by a system-trusted source indicates whether or not the object is digitally signed by a system-trusted source. If *signed by a system-trusted source* is hex 01, *sign state* will also be hex 01.

The **object authority list status** field indicates whether or not the object is contained in an authority list. If it is, the **authority list identification information** provides the name of the authority list, except when the authority list is indicated as destroyed, in which case, the name information is meaningless.

The **dump for previous release reason code** can be used to determine why the object is not eligible according to the *dump for previous release permitted* field. Currently reason codes are architected for programs and for modules.

The reason codes are mapped as follows for program objects. Note that more than one reason may be returned.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Program dump for previous release reason codes	Char(8)	
0	0		Language version, release, and modification level reason	Bit 0
			0 = Language version, release, and modification level is not a reason	
			1 = Language version, release, and modification level is one reason	
0	0		Level of machine interface used reason	Bit 1

Offset Dec	Offset Hex	Field Name	Data Type and Length
			0 = The level of machine interface used is not a reason **1 =** The level of machine interface used not available in the previous release
0	0		Program observability reason Bit 2 **0 =** Lack of program observability is not a reason **1 =** Program is not observable and must be to be moved to previous release
0	0		Program compressed reason Bit 3 **0 =** Program compression is not a reason **1 =** The program, of type *non-bound program*, is compressed and the previous release does not support compression of the *non-bound program* type of program **Note:** This reason code does not apply to bound programs, bound service programs, or Java programs.
0	0		Bound program or bound service program reason Bit 4 **0 =** The program type is not a reason **1 =** Bound programs and bound service programs are not supported on the previous release
0	0		Bound program or bound service program retranslation reason Bit 5 **0 =** Retranslation is not a reason **1 =** Retranslation of bound programs and bound service programs are not supported on the previous release
0	0		Java program reason Bit 6 **0 =** The program type is not a reason **1 =** Java programs are not supported on the previous release
0	0		Target version, release, and modification level reason Bit 7 **0 =** Target version, release, and modification level is not a reason **1 =** Target version, release, and modification level is a reason **Note:** This reason code applies only to programs whose target version, release, and modification level is hex 0510 or later.
0	0		Reserved Bits 8-63
8	8	—- End —-	

The reason codes are mapped as follows for module objects. Note that more than one reason may be returned.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Module dump for previous release reason codes	Char(8)	
0	0		Language version, release, and modification level reason	Bit 0
			0 = Language version, release, and modification level is not a reason	
			1 = Language version, release, and modification level is one reason	
0	0		Level of machine interface used reason	Bit 1
			0 = The level of machine interface used is not a reason	
			1 = The level of machine interface used not available in the previous release	
0	0		Module observability reason	Bit 2
			0 = Lack of module observability is not a reason	
			1 = Module is not observable and must be moved to previous release	
0	0		Module reason	Bit 3
			0 = The object type of module is not a reason	
			1 = Modules are not supported on the previous release	
0	0		Module retranslation reason	Bit 4
			0 = Retranslation is not a reason	
			1 = Retranslation of modules is not supported on the previous release	
0	0		Target version, release, and modification level reason	Bit 5
			0 = Target version, release, and modification level is not a reason	
			1 = Target version, release, and modification level is a reason	
			Note: This reason code applies only to modules whose target version, release, and modification level is hex 0510 or later.	
0	0		Reserved	Bits 6-63
8	8	—- End —-		

The **timestamp of last use of object** field and the **count of number of days object was used** field are set by the Call External (CALLX) or Transfer Control (XCTL) instructions on the objects first use on that day. The timestamp value is only good for the date. The time value obtained from this timestamp is not accurate.

The **type of program** field indicates the program model of a program object, which is determined by how the program was created. It is only present when operand 2 points to a program object. This field is

necessary since the object type and object subtype do not provide enough information to identify the program model of a program object. Knowing the program type is useful in selecting appropriate program specific instructions.

The **domain of object** field contains the value of the state under which a program or procedure must be running to access this object.

The **state for program or module** field contains the state under which the program runs. It is only present when the **state provided** flag is on.

The **MI-supplied information** is simply an 8 byte character field which can be set into an object and materialized with the Materialize System Object (MATSOBJ) instruction. The machine has no knowledge or dependencies on the content of this field.

The **earliest compatible release** field contains the earliest release in which the object can be used.

The **primary group identification** field contains the type, subtype, and name of the primary group profile for the object. If the primary group for the object is not set, the type field will be set to a value of hex 00.

The **hardware storage protection** field contains the hardware storage protection of the object's encapsulated part, of it's associated space and the protection level requested when the object was created. Each 2-bit subfield contains a value whose meaning is defined as follows:

00 =	Reference and modify allowed for user state programs
01 =	Only reference allowed for user state programs
10 =	Only reference allowed for all programs
11 =	No reference nor modify allowed for user state programs

The last two subfields indicate when hardware storage protection is enforced.

The **file identifier**, in combination with the *generation ID*, uniquely identifies the object within a file system. The *file ID* may be reused when the object is deleted.

The **generation identifier** uniquely identifies the generation of the usage of a file identifier. The *generation ID* for a given *file ID* is not reused,

If the object has not been assigned a *file ID* and *generation ID*, binary 0s will be returned.

If the object specified by operand 2 is attached to a byte stream file or a Composite Object Group (parent object), the **parent of attached object** field returns a system pointer to the byte stream file or Composite Object Group. Otherwise, a null pointer value is returned.

Note:	Referencing this pointer requires that the *receiver* (operand 1) be 16-byte aligned. The machine enforces only 4-byte alignment of the *receiver*.

Number of signers indicates the number of digital certificates currently having signed the object.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

  –

  – Contexts referenced for address resolution

Lock Enforcement

-
- Materialize
  -
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Footnotes:

[1] 'Previous release' refers to the previous mandatory release. This is release N-1, mod level zero when release N is the current release. (For version 2, release 1.1, the previous mandatory release is version 1, release 3.0.).

Materialize Machine Data (MATMDATA)

Op Code (Hex)	Operand 1	Operand 2
0522	Receiver	Materialization option

Operand 1: Character variable scalar.

Operand 2: Character(2) constant, or unsigned binary(2) constant or immediate.

Bound program access
Built-in number for MATMDATA is 160. MATMDATA (receiver : address materialization_option : literal(2) OR literal(4)) The *materialization_option* may be declared as a literal of any scalar data type. **-- OR --** Built-in number for MATTOD is 94. MATTOD (time_of_day : address) The time-of-day clock is materialized. This function is identical to MATMDATA when a *materialization_option* value of Hex 0000 is specified.

Description: The machine data requested by *materialization option* is returned at the location specified by *receiver*. For the purposes of this instruction, machine data refers to any data that is encapsulated by the machine. The data can be either thread-specific or apply system-wide.

Operand 2 is a 2-byte value. The value of operand 2 determines which machine data are materialized. Operand 2 is restricted to a constant character or unsigned binary scalar or an immediate value. A summary of the allowable values for Operand 2 follows.

Table 1. Materialization option

Option value	Description	Page
Hex 0000	Materialize time-of-day clock as local time	″Hex 0000 = Materialize time-of-day clock as local time″ (page)
Hex 0001	Materialize system parameter integrity validation flag	″Hex 0001 = Materialize system parameter integrity validation flag″ (page)
Hex 0002	Materialize thread execution mode flag	″Hex 0002 = Materialize thread execution mode flag″ (page)
Hex 0003	Materialize maximum size of a space object or associated space when space alignment is chosen by the machine	″Hex 0003 = Materialize maximum size of a space object or associated space when space alignment is chosen by the machine″ (page)
Hex 0004	Materialize time-of-day clock as Coordinated Universal Time (UTC)	″Hex 0004 = Materialize time-of-day clock as Coordinated Universal Time (UTC)″ (page)
Hex 0005 though FFFF	Reserved	

Operand 1 specifies a *receiver* into which the materialized data is placed. It must specify a character scalar with a minimum length which is dependent upon the *materialization option* specified for operand 2. The *receiver* may be substringed. The start position of the substring may be a variable. However, the length of the substring must be an immediate or constant. The length specified for operand 1 must be at least the required minimum. Only the bytes up to the required minimum length are used. Any excess bytes are ignored.

The data placed into the *receiver* differs depending upon the *materialization option* specified. The following descriptions detail the formats of the optional materializations.

Hex 0000 = Materialize time-of-day clock as local time: (minimum *receiver* length is 8)

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Time of day	Char(8)
8	8	—- End —-	

Time of day is the time value of the time-of-day clock which is returned as the local time for the system. See "Standard Time Format" on page 1272 for a detailed description of the format for a time value.

Unpredictable results occur if the time-of-day clock is materialized before it is set.

The time-of-day clock can be materialized as the Coordinated Universal Time (UTC) for the system using ″Hex 0004 = Materialize time-of-day clock as Coordinated Universal Time (UTC)″ (page).

See "Time-of-Day (TOD) Clock" on page 1273 for detailed descriptions of the time-of-day clock, local time, and UTC.

Performance note: The time-of-day clock may be materialized, with the *time of day* returned as the local time for the system, with this instruction and also with the Materialize Machine Attributes (MATMATR) instruction. Better performance may be realized with the use of this instruction rather than with the MATMATR instruction.

Hex 0001 = Materialize system parameter integrity validation flag: (minimum *receiver* length is 1)

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	System parameter integrity validation flag	Char(1)
1	1	—- End —-	

This option returns the value of the machine attribute which specifies whether additional validation of parameters passed to programs which run when the thread is in system state is to be performed, such as for U. S. government's Department of Defense security ratings.

A value of hex 01 indicates this additional checking is being performed. A value of hex 00 is returned otherwise.

Hex 0002 = Materialize thread execution mode flag: (minimum *receiver* length is 1)

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Thread execution mode flag	Char(1)
1	1	—- End —-	

This option returns the value of the thread execution mode for the thread in which the instruction is run.

A returned value of hex 01 indicates that thread is currently executing in kernel mode. A value of hex 00 is returned otherwise.

Hex 0003 = Materialize maximum size of a space object or associated space when space alignment is chosen by the machine: (minimum *receiver* length is 4)

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Maximum size of machine-aligned space object or associated space	UBin(4)
4	4	—- End —-	

This option returns the maximum size in bytes of a space object or associated space created with the space alignment chosen by the machine. Some types of objects may not support an associated space of the maximum size.

This size may vary with each machine implementation.

Hex 0004 = Materialize time-of-day clock as Coordinated Universal Time (UTC): (minimum *receiver* length is 8)

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Time of day	Char(8)
8	8	—- End —-	

Time of day is the time value of the time-of-day clock which is returned as the Coordinated Universal Time (UTC) for the system. See "Standard Time Format" on page 1272 for a detailed description of the format for a time value.

Unpredictable results occur if the time-of-day clock is materialized before it is set.

The time-of-day clock can be materialized as the local time for the system using "Hex 0000 = Materialize time-of-day clock as local time" (page 694).

See "Time-of-Day (TOD) Clock" on page 1273 for detailed descriptions of the time-of-day clock, local time, and UTC.

Performance note: The time-of-day clock may be materialized, with the *time of day* returned as the UTC for the system, with this instruction and also with the Materialize Time Of Day Attributes (MATTODAT) instruction. Better performance may be realized with the use of this instruction rather than with the MATTODAT instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize Time of Day Clock Attributes (MATTODAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0553	Materialization template	Attribute selection	Result

Operand 1: Space pointer.

Operand 2: Unsigned binary(4) scalar.

Operand 3: Signed binary(4) variable scalar.

Bound program access

```
Built-in number for MATTODAT is 666.
MATTODAT (
          materialization_template : address
          attribute_selection      : unsigned binary(4)
) :   signed binary(4)  /* result */
```

Description: The time-of-day clock attribute specified by *attribute selection* for the current system is materialized as specified in the *materialization template*. Upon successful completion, *result* is set to binary 0.

The *materialization template* must be 16-byte aligned. If not, the EFAULT error is returned in *result*. The *materialization template* has the following format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Template size	Char(8)	
0	0		Number of bytes provided by the user	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Attribute specifications as defined by the attribute selection operand	Char(*)	
*	*	—- End —-		

The first 4 bytes of the *materialization template* identify the total **number of bytes provided by the user** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the ENOSPC error to be returned in *result*.

The second 4 bytes of the *materialization template* identify the total **number of bytes available for materialization**. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested for materialization, then the excess bytes are unchanged. No errors (other than the ENOSPC error described previously) are returned in the event that the receiver contains insufficient area for the materialization.

A summary of the allowable values for *attribute selection* follows. If an invalid *attribute selection* value is specified, the EINVAL error number is returned in *result*.

Table 1. MATTODAT selection values

Selection value	Description	Page
1	Time-of-day clock as Coordinated Universal Time (UTC)	″1 = Time-of-day clock as Coordinated Universal Time (UTC)″ (page 939)
2	Time-of-day clock adjustment	″2 = Time-of-day clock adjustment″ (page 939)

1 = Time-of-day clock as Coordinated Universal Time (UTC):

Offset Dec	Offset Hex	Field Name	Data Type and Length
8	8	Reserved (binary 0)	Char(8)
16	10	Time of day	Char(8)
24	18	Time zone offset	Bin(4)
28	1C	Reserved (binary 0)	Char(4)
32	20	—- End —-	

Time of day is the time value of the time-of-day clock which is returned as the Coordinated Universal Time (UTC) for the system. See “Standard Time Format” on page 1272 for a detailed description of the format for a time value.

Time zone offset indicates the local time zone, including any adjustment for Daylight Savings Time, as measured in minutes of time westward from Greenwich, England.

Unpredictable results occur if the time-of-day clock is materialized before it is set.

See “Time-of-Day (TOD) Clock” on page 1273 for detailed descriptions of the time-of-day clock, UTC, time zone offset, and local time.

2 = Time-of-day clock adjustment:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
8	8	Adjustment options	Char(2)		
8	8		Adjustment status		Bit 0
			0 =	Time-of-day clock adjustment not active	
			1 =	Time-of-day clock adjustment active	
8	8		Adjustment direction		Bit 1
			0 =	Increase time of day	
			1 =	Decrease time of day	
8	8		Reserved (binary 0)		Bits 2-15
10	A	Reserved (binary 0)	Char(6)		
16	10	Time interval	Char(8)		
24	18	Adjustment duration	Char(8)		
32	20	—- End —-			

Time-of-day clock adjustment is not supported on all hardware levels. If time-of-day clock adjustment is not supported, then the ENOTSUP error is returned in *result*.

Adjustment status indicates whether or not a time-of-day clock adjustment is active for the system. When the *adjustment status* is binary 0, then a time-of-day clock adjustment is not active and the *adjustment direction*, *time interval*, and *adjustment duration* fields will be set to binary 0. When the *adjustment status* is binary 1, then a time-of-day clock adjustment is active and the *adjustment direction*, *time interval*, and *adjustment duration* fields will indicate the status of the active adjustment.

Adjustment direction indicates the direction of the time-of-day clock adjustment for the system. When the *adjustment direction* is binary 0, the rate at which the time-of-day clock runs is increased until the adjustment is completed. When the *adjustment direction* is binary 1, the rate at which the time-of-day clock runs is decreased until the adjustment is completed.

Time interval is a time value which specifies the remaining amount of time by which the time-of-day clock will be increased or decreased. See "Standard Time Format" on page 1272 for a detailed description of the format for a time value.

Adjustment duration is a time value which provides an estimate of the amount of time required in order to complete the time-of-day clock adjustment. See "Standard Time Format" on page 1272 for a detailed description of the format for a time value.

An active time-of-day clock adjustment results in small adjustments to the time of day such that upon completion of the adjustment the time of day has increased or decreased by the amount specified in the *time interval* field. If the adjustment is to increase the time of day, the adjustment is made by increasing the rate at which the time-of-day clock is incremented. To decrease the time of day, the time-of-day clock is incremented more slowly than normal. When the adjustment has been completed, the rate at which the time-of-day clock is incremented returns to normal. The time of day continues to be a monotonically increasing value while an adjustment is active.

A time-of-day clock adjustment will remain active until completed unless one of the following occur:

-
- A new time-of-day clock adjustment is started for the system
- The time-of-day clock for the system is modified
- The system is powered off.

Warning: Temporary Level 3 Header

Authorization Required

-
- None

Lock Enforcement

-
- None

Error conditions

The *result* will be set to one of the following:

EFAULT	3408 - The address used for an argument was not correct. The template was not aligned on the required boundary or the storage was inaccessible.

EINVAL 3021 - The value specified for the argument is not correct.

ENOSPC 3404 - No space available.

ENOTSUP 3440 - Operation not supported.

Time-of-day clock adjustment is not supported for all hardware levels.

Exceptions

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

Materialize User Profile (MATUP)

Op Code (Hex)	Operand 1	Operand 2
013E	Receiver	User profile

Operand 1: Space pointer.

Operand 2: System pointer or space pointer data object.

Bound program access

```
Built-in number for MATUP is 62.
MATUP (
       receiver       : address
       user_profile   : address of system pointer OR
                        address of space pointer(16)
)
```

Description: The attributes of the *user profile* specified by operand 2 are materialized into the *receiver* specified by operand 1. (Operand 2 may refer to a *materialization template* that contains a system pointer to the user profile.)

The *receiver* identified by operand 1 must be 16-byte aligned in the space. The following is the format of the materialized information:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Object identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Object creation options	Char(4)	
40	28		Existence attribute	Bit 0

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			1 = Permanent	
40	28		Space attribute	Bit 1
			0 = Fixed-length	
			1 = Variable-length	
40	28		Reserved (binary 1)	Bit 2
40	28		Reserved (binary 0)	Bits 3-12
40	28		Initialize space	Bit 13
40	28		Reserved (binary 0)	Bits 14-31
44	2C	Reserved (binary 0)	Char(4)	
48	30	Size of space	Bin(4)	
52	34	Initial value of space	Char(1)	
53	35	Performance class	Char(4)	
57	39	Reserved (binary 0)	Char(7)	
64	40	Reserved (binary 0)	Char(16)	
80	50	Reserved (binary 0)	Char(16)	
96	60	Privileged instructions (1 = authorized)	Char(4)	
96	60		Create Logical Unit Description	Bit 0
96	60		Create Network Description	Bit 1
96	60		Create Controller Description	Bit 2
96	60		Create User Profile	Bit 3
96	60		Modify User Profile	Bit 4
96	60		Diagnose	Bit 5
96	60		Terminate Machine Processing	Bit 6
96	60		Initiate Process	Bit 7
96	60		Modify Resource Management Controls	Bit 8
96	60		Create Mode Description	Bit 9
96	60		Create Class of Service Description	Bit 10
96	60		Reserved (binary 0)	Bits 11-31
100	64	Special authorizations (1 = authorized)	Char(4)	
100	64		All object authority	Bit 0
100	64		Load (unrestricted)	Bit 1
100	64		Dump (unrestricted)	Bit 2
100	64		Suspend object (unrestricted)	Bit 3
100	64		Load (restricted)	Bit 4
100	64		Dump (restricted)	Bit 5
100	64		Suspend object (restricted)	Bit 6
100	64		Process control	Bit 7
100	64		Reserved (binary 0)	Bit 8
100	64		Service authority	Bit 9
100	64		Auditor authority	Bit 10
100	64		Spool control	Bit 11
100	64		I/O system configuration	Bit 12
100	64		Reserved (binary 0)	Bits 13-23

Offset Dec	Hex	Field Name	Data Type and Length	
100	64		Modify machine attributes	Bits 24-31
100	64		Group 2	Bit 24
100	64		Group 3	Bit 25
100	64		Group 4	Bit 26
100	64		Group 5	Bit 27
100	64		Group 6	Bit 28
100	64		Group 7	Bit 29
100	64		Group 8	Bit 30
100	64		Group 9	Bit 31
			Note: Group 1 requires no authorization.	
104	68	System storage authorization	Bin(4)	
		The maximum amount of auxiliary storage (in units of 1,024 bytes) that can be allocated for the storage of objects owned by this user profile in the system ASP and basic ASPs		
108	6C	System storage utilization	Bin(4)	
112	70	User profile status	Char(2)	
112	70		Verify storage utilization	Bit 0
			0 = Storage utilization has been verified and is correct	
			1 = Storage utilization has not been verified and may not be correct	
112	70		Reserved (binary 0)	Bits 1-15
114	72	Identification flags	Char(1)	
114	72		Reserved (binary 0)	Bits 0-1
114	72		User identification specified	Bit 2
114	72		Group identification specified	Bit 3
114	72		Reserved (binary 0)	Bits 4-7
115	73	Object audit level	Char(1)	
115	73		Reserved (binary 0)	Bits 0-5
115	73		Audit object changes for this user	Bit 6
115	73		Audit object reads for this user	Bit 7
116	74	User audit level 1	Char(4)	
116	74		Operating system defined	Bits 0-1
116	74		Security function auditing	Bit 2
			0 = Security function auditing is not active	
			1 = Security function auditing is active	
116	74		Operating system defined	Bits 3-6
116	74		Signal action auditing	Bit 7
			0 = Signal action auditing is not active	
			1 = Signal action auditing is active	

Offset Dec	Offset Hex	Field Name	Data Type and Length		
116	74		Operating system defined	Bits 8-31	
120	78	User audit level 2	Char(4)		
120	78		Program adoption auditing	Bit 0	
			0 = Program adoption auditing is not active		
			1 = Program adoption auditing is active		
120	78		Reserved (binary 0)	Bits 1-31	
124	7C	User identification	UBin(4)		
128	80	Group identification	UBin(4)		
132	84	Number of independent ASP entries	UBin(2)		
134	86	Reserved (binary 0)	Char(10)		
144	90	Counts of profile entries	Char(64)		
144	90		Ownership entries	Char(8)	
144	90		Number of used entries		UBin(4)
148	94		Number of possible available entries		UBin(4)
152	98		Authorization entries	Char(8)	
152	98		Number of used entries		UBin(4)
156	9C		Number of possible available entries		UBin(4)
160	A0		Authorized user entries	Char(8)	
160	A0		Number of used entries		UBin(4)
164	A4		Number of possible available entries		UBin(4)
168	A8		Primary group entries	Char(8)	
168	A8		Number of used entries		UBin(4)
172	AC		Number of possible available entries		UBin(4)
176	B0		Reserved (binary 0)	Char(32)	
208	D0	Reserved (binary 0)	Char(8)		
216	D8	Total storage utilization	Char(8)		
224	E0	Individual independent ASP information	[*] Char(16)		
		(repeated once for each independent ASP, up to the number of independent ASPs supported (223))			
224	E0		Independent ASP storage authorization	Bin(4)	
228	E4		Independent ASP storage utilization	Bin(4)	
232	E8		Reserved (binary 0)	Char(2)	
234	EA		Specification flags	Char(1)	
234	EA		Independent ASP storage authorization setting		Bit 0
			0 = The value of *independent ASP storage authorization* is a default value.		
			1 = The value of *independent ASP storage authorization* was set to a specific value by MODUP or CRTUP.		
234	EA		Reserved (binary 0)		Bits 1-7
235	EB		Status flags	Char(1)	
235	EB		User profile extension existence flag		Bit 0

Offset Dec	Hex	Field Name	Data Type and Length		
			0 =	This independent ASP does not have a user profile extension for this user profile.	
			1 =	This independent ASP has a user profile extension for this user profile	
235	EB		User profile extension damaged		Bit 1
			0 =	The user profile extension is not damaged	
			1 =	The user profile extension is damaged	
235	EB		Reserved (binary 0)		Bits 2-7
236	EC		Reserved (binary 0)		Char(4)
*	*	—- End —-			

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the *receiver*. If the byte area identified by the *receiver* is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the *receiver* contains insufficient area for the materialization and the receiver may contain a partial *individual independent ASP information* entry.

System storage utilization is the current amount of auxiliary storage (in units of 1,024 bytes) allocated for the storage of objects owned by this user profile in the system ASP and basic ASPs.

The **verify storage utilization** field returns either binary 0 or 1. If binary 0 is returned, the value returned in the *total storage utilization* field has been verified and it is correct. That is, all values returned in the *system storage utilization* and *independent ASP storage utilization* for varied-on independent ASPs have been verified and they are correct. If binary 1 is returned, the value returned in the *total storage utilization* field has not been verified and may not be correct. That is, at least one of the values returned in *system storage utilization* and *independent ASP storage utilization* for varied on independent ASPs have not been verified and may not be correct. The MATAUOBJ instruction can be used to correct all the values of storage utilization. After MATAUOBJ is issued to correct the storage utilization values, the MATUP instruction must be issued again to retrieve the corrected values.

Note: The storage utilization of independent ASPs that are not varied on is not verified.

The **security function auditing** field specifies whether an audit record is created for identified security-related functions. (See specific instructions for which security-related functions an audit record is to be created.)

The **signal action auditing** field specifies whether an audit record is created when a signal, that is not ignored, is delivered to a thread.

The **program adoption auditing** field specifies whether an audit record is created when authority is obtained through program adoption.

The **number of independent ASP entries** field is the total number of *individual independent ASP information* entries. 223 is returned even if *number of bytes provided* is not large enough to contain all of the *individual independent ASP information* entries.

Ownership entries refer to the entries created in the user profile due to owned objects.

Authorization entries refer to the entries created in the user profile due to objects to which this profile has been authorized.

Authorized user entries refer to the entries created in the user profile due to owned objects which have been authorized to other user profiles.

Primary group entries refer to the entries created in the user profile due to objects for which this profile is the primary group.

Number of used entries is the number of entries currently in the user profile for the specified type of entry.

Number of possible available entries is the number of entries which can possibly be added to the user profile.

The **total storage utilization** (in units of 1,024 bytes) field contains the sum of the system *storage utilization* (storage used on the system and basic ASPs) and all the *independent ASP storage utilization* of independent ASPs that are currently varied on.

The **individual independent ASP information** is an array of information for the individual independent ASPs. This array starts from the independent ASP 33 and increases sequentially to 255. As many entries as will fit in *number of bytes provided* will be returned up to a maximum of *number of independent ASP entries*. It is possible for a partial entry to be returned.

The **independent ASP storage authorization** field contains the maximum amount of independent auxiliary storage (in units of 1,024 bytes) that can be owned by this user profile. A value will always be returned in this field, even for an independent ASP that is varied off.

The **independent ASP storage utilization** field contains the current amount of independent auxiliary storage (in units of 1,024 bytes) allocated for the permanent objects owned by this user profile if the independent ASP is varied on. If the independent ASP is not varied on, 0 is returned.

The **user profile extension existence flag** field indicates whether or not the independent ASP has a user profile extension for this user profile. A user profile extension is stored on the independent ASP and contains information about the user profile.

If operand 2 is a system pointer, it identifies the input user profile object. If operand 2 is a space pointer, it provides addressability to the **materialization template**. The *materialization template* must be aligned on a 16-byte boundary. The format of the template is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization template	Char(64)	
0	0		Template version	Char(1)
1	1		Reserved (binary 0)	Char(1)
2	2		Materialization options	Char(2)
2	2		Materialize for given release	Bit 0

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = Materialize data applicable to the current release	
			1 = Materialize data applicable to the specified release	
2	2		Reserved (binary 0)	Bits 1-15
4	4		Reserved (binary 0)	Char(10)
14	E		Target release to materialize	Char(2)
14	E		Reserved (binary 0)	Bits 0-3
14	E		Version	Bits 4-7
14	E		Release	Bits 8-11
14	E		Modification level	Bits 12-15
16	10		User profile	System pointer
32	20		Reserved (binary 0)	Char(32)
64	40	—- End —-		

The **template version** identifies the version of the materialization template. It must be set to hex 00.

The **materialization options** are options that control how the user profile data is presented in the *receiver*.

If the **materialize for given release** bit is set to binary 1, then the operand 1 *receiver* will be filled in with values applicable to the release specified by the *target release to materialize* field. Fields that are not defined in the target release will be returned as hex zeroes.

The **target release to materialize** field is the target release specified when the *materialize for given release* bit is set to binary 1. The earliest release that can be specified is Version 5, Release 1, Modification level 0. The latest release that can be specified is the current release. If an invalid release is specified, a *template value invalid* (hex 3801) exception is signaled.

The **user profile** system pointer is the user profile to be materialized.

Warning: Temporary Level 3 Header

Authorization Required

•

• Operational

  –

  – User profile identified by operand 2

Lock Enforcement

•

• Materialize

  –

  – User profile identified by operand 2

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Materialize User Profile Pointers from ID (MATUPID)

Op Code (Hex)	Operand 1	Operand 2
013A	Return template	Input template

Operand 1: Space pointer.

Operand 2: Space pointer.

Bound program access

```
Built-in number for MATUPID is 382.
MATUPID (
         return_template   : address
         input_template    : address
)
```

Description: This instruction converts the uids and/or gids specified by the *input template* in operand 2 to system pointers for the corresponding user profiles returned in operand 1. The materialization options determine the format of the profile information returned in operand 1.

The operand 2 template must be on a 4-byte boundary. The format of the *input template* is:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Materialization format option **Hex 01 =** Materialize short template **Hex 02 =** Materialize long template	Char(1)
1	1	Materialization type option **Hex 00 =** List provided **Hex 41 =** Return all gids starting with the specified gid **Hex 80 =** Return all uids then all gids **Hex 81 =** Return all uids then gids starting with the specified uid	Char(1)
2	2	Number of uids provided	UBin(4)
6	6	Number of gids provided	UBin(4)
10	A	Reserved (binary 0)	Char(10)
20	14	—- End —-	

The **materialization format option** field identifies the format of the materialization to be returned in the *return template* specified by operand 1.

The **materialization type option** field identifies the type of materialization operation to be performed. When one of the following is not specified, a *template value invalid* (hex 3801) exception is signalled.

•

- *List provided* - a list of uids and/or gids is provided (number of entries is indicated by *number of uids provided* and *number of gids provided*). An entry for each uid and gid in the list will be materialized into operand 1.
- *Return all uids then gids* - return all uids, then gids starting with the first uid. Only as many uids and gids will be returned as will fit in the space provided for operand 1.
- *Return all uids then gids starting with the specified uid* - return all uids/gids beginning with the uid specified. An entry for each uid and gid starting with the uid specified will be materialized into operand 1. If the specified uid is not found, then the next entry found in the table will be returned as the first entry in the materialization template. The next entry found will either be the next uid in numerical order, or the first gid.
- *Return all gids starting with the specified gid* - return all gids beginning with the gid specified. An entry for each gid starting with the gid specified will be materialized into operand 1. If the specified gid is not found, then the next entry found in the table will be returned as the first entry in the materialization template. The next entry found will be the next gid in numerical order. To get a list of all gids start with a gid of 1 (hex 00000001).

  Note: Starting with a specified uid/gid does not guarantee that the list has not changed. When trying to get all uids/gids and a continuation is required, any ID's added between calls that are less than the uid/gid specified as the resume will not be returned.

The **number of uids provided** field is the number of uids in the list of uids that follows the reserved area. This may be zero (no uids) or a positive number. This field is used for *materialization type option* hex 00; otherwise ignored.

The **number of gids provided** field is the number of gids in the list of gids that follows any uids provided after the reserved area. This may be zero (no gids) or a positive number. This field is used for *materialization type option* hex 00; otherwise ignored.

This information will be followed by a list of uids specified. If *materialization type option* hex 81 is selected, one uid will be in the list. Each uid has the following format.

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Uid	UBin(4)
4	4	—- End —-	

Following the uids will be a list of gids specified. If *materialization type option* hex 41 is selected, one gid will be in the list. Each gid has the following format.

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Gid	UBin(4)
4	4	—- End —-	

The receiver identified by operand 1 must be 16-byte aligned in the space. The format of the materialization is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization	Bin(4)
8	8	Number of uids returned	UBin(4)	
12	C	Number of gids returned	UBin(4)	
16	10	Indicators	Char(1)	
16	10		Pointer(s) not set	Bit 0
16	10		Reserved	Bits 1-7
17	11	Reserved	Char(15)	
32	20	—- End —-		

The **number of bytes provided for materialization** identify the total number of bytes that may be used by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The **number of bytes available for materialization** identify the total number of bytes available to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

The **number of uids returned** field contains the total number of uids materialized.

The **number of gids returned** field contains the total number of gids materialized.

Pointer(s) not set. When this field is binary 1, one or more *user profile* instances in the uid and/or gid list is binary 0s because a uid or gid in the list provided is not in use or the user profile associated with the uid/gid is destroyed.

This information will be followed by an entry for each uid in the list in operand 2 followed by an entry for each gid in the list in operand 2.

For the short format, each entry (for uids and gids) in the list will have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	User profile	System pointer
16	10	—- End —-	

For the long format, each entry (for uids and gids) in the list will have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	User profile type code	Char(1)	
1	1	User profile subtype code	Char(1)	
2	2	User profile name	Char(30)	
32	20	Uid/gid	UBin(4)	
36	24	ID type	Char(1)	
		Hex 01 = uid		
		Hex 02 = gid		
37	25	Flags	Char(1)	
37	25		User profile pointer is not set	Bit 0
37	25		Reserved	Bits 1-7
38	26	Reserved	Char(10)	
48	30	User profile	System pointer	
64	40	—- End —-		

User profile pointer is not set. When this field is binary 1, the uid or gid specified is not in use or the user profile is destroyed. *User profile* will be set to binary 0s.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A0A ID Index Not Available

10 Damage Encountered

1002 Machine Context Damage State

1005 Authority Verification Terminated Due to Damaged Object

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Memory Compare (MEMCMP)

Bound program access

```
Built-in number for MEMCMP is 17.
MEMCMP (
        string1         : address of aggregate(*)
        string2         : address of aggregate(*)
        string_length   : unsigned binary(4) value which specifies the
                          length of string1 and string2
) :   signed binary(4) value which indicates if string1 is lexically
      less than (-1), equal to (0) or greater than (1) string2
```

Description: A compare is done of the storage specified by *string1* and *string2*. If the first byte of *string1* is less than the first byte of *string2*, the function returns -1; if the byte is greater the function returns 1. If the bytes are equal the function continues with the next byte. This process is repeated until the number of bytes specified by *string length* have been compared. If all bytes compare equal, the function returns 0. If the *string length* value is zero, the function also returns 0.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Memory Copy (MEMCPY)

Bound program access
Built-in number for MEMCPY is 15. MEMCPY (target_string : address of aggregate(*) source_string : address of aggregate(*) copy_length : unsigned binary(4) value which specifies the number of bytes to copy) : space pointer(16) to the target string

Description: A copy from the storage specified by *source string* to the storage specified by *target string* is performed. *Copy length* specifies the number of bytes to copy. It is assumed that sufficient storage exists at the *target string* location to receive the specified number of bytes.

Pointers can be copied using this function. However, the source and target strings must be like-aligned.

Undefined results can occur if the storage locations specified by *target string* and *source string* overlap.

Copy Bytes with Pointers (CPYBWP) should be used when possible for performance reasons.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Memory Move (MEMMOVE)

Bound program access

```
Built-in number for MEMMOVE is 16.
MEMMOVE (
         target_string   : address of aggregate(*)
         source_string   : address of aggregate(*)
         copy_length     : unsigned binary(4) value which specifies the
                           number of bytes to copy
) :   space pointer(16) to the target string
```

Description: A copy from the storage specified by *source string* to the storage specified by *target string* is performed. *Copy length* specifies the number of bytes to copy. It is assumed that sufficient storage exists at the *target string* location to receive the specified number of bytes.

Pointers can be copied using this function. However, the source and target strings must be like-aligned.

Results are defined if the storage locations specified by *target string* and *source string* overlap. The result is equivalent to copying the *source string* first to a temporary location and then from the temporary location to the *target string.*

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Modify Automatic Storage Allocation (MODASA)

Op Code (Hex)	Operand 1	Operand 2
02F2	Storage allocation	Modification size

Operand 1: Space pointer data object or null.

Operand 2: Signed binary scalar.

Bound program access

```
Built-in number for MODASA is 159.
MODASA (
        modification_size   : signed binary(4) OR
                              unsigned binary(4)
) :   space pointer(16) to a storage allocation
```

where 0 < *modification size* <= 16,773,119.

```
-- OR --

Built-in number for MODASA2 is 397.
MX translator only

MODASA2 (
         modification_size   : signed binary(4) OR
                               unsigned binary(4)
) :   space pointer(16) to a storage allocation
```

where -16,773,119 <= *modification size* <= 16,773,119 for signed binary values and 0 <= *modification size* <= 16,773,119 for unsigned binary values.

The *modification size* operand corresponds to operand 2 on the MODASA operation; the return value corresponds to operand 1.

Description: The automatic storage frame (ASF) of the current invocation is extended or truncated by the *modification size* specified by operand 2. A positive value indicates that the frame is to be extended; a negative value indicates that the frame is to be truncated; a zero value does not change the ASF. If operand 1 is not null, it will be treated as follows:

•

- ASF extension: receives the address of the first byte of the extension. The ASF extension might not be contiguous with the remainder of the ASF allocation.

- ASF truncation: operand 1 should be null for truncation. If operand 1 is not null, then addressability to the first byte of the deallocated space is returned. This value should not be used as a space pointer since it locates space that has been deallocated.
- If a value of zero is specified for operand 2: the value returned is unpredictable.

When the ASF is extended, the extension is aligned on a 16-byte boundary. An extension is not initialized.

A *scalar value invalid* (hex 3203) exception is signaled if the truncation amount would include the storage for all automatic data objects for the current invocation, including the initial allocation.

A space pointer machine object cannot be specified for operand 1.

Warning: Temporary Level 3 Header

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

16 Exception Management

1604 Retry/Resume Invalid

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C1D Automatic Storage Overflow

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Modify Automatic Storage Allocation (MODASA)

Op Code (Hex)	Operand 1	Operand 2
02F2	Storage allocation	Modification size

Operand 1: Space pointer data object or null.

Operand 2: Signed binary scalar.

Bound program access
Built-in number for MODASA is 159. MODASA (modification_size : signed binary(4) OR unsigned binary(4)) : space pointer(16) to a storage allocation where 0 < *modification size* <= 16,773,119. **-- OR --** Built-in number for MODASA2 is 397. **MX translator only** MODASA2 (modification_size : signed binary(4) OR unsigned binary(4)) : space pointer(16) to a storage allocation where -16,773,119 <= *modification size* <= 16,773,119 for signed binary values and 0 <= *modification size* <= 16,773,119 for unsigned binary values. The *modification size* operand corresponds to operand 2 on the MODASA operation; the return value corresponds to operand 1.

Description: The automatic storage frame (ASF) of the current invocation is extended or truncated by the *modification size* specified by operand 2. A positive value indicates that the frame is to be extended; a negative value indicates that the frame is to be truncated; a zero value does not change the ASF. If operand 1 is not null, it will be treated as follows:

•

- ASF extension: receives the address of the first byte of the extension. The ASF extension might not be contiguous with the remainder of the ASF allocation.
- ASF truncation: operand 1 should be null for truncation. If operand 1 is not null, then addressability to the first byte of the deallocated space is returned. This value should not be used as a space pointer since it locates space that has been deallocated.
- If a value of zero is specified for operand 2: the value returned is unpredictable.

When the ASF is extended, the extension is aligned on a 16-byte boundary. An extension is not initialized.

A *scalar value invalid* (hex 3203) exception is signaled if the truncation amount would include the storage for all automatic data objects for the current invocation, including the initial allocation.

A space pointer machine object cannot be specified for operand 1.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

16 Exception Management

- 1604 Retry/Resume Invalid

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2C Program Execution

- 2C1D Automatic Storage Overflow

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Modify Exception Description (MODEXCPD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03EF	Exception description	Modifying attributes	Modification option

Operand 1: Exception description.

Operand 2: Space pointer or character(2) constant.

Operand 3: Character(1) scalar.

Description: The exception description attributes specified by operand 3 are modified with the values of operand 2.

Operand 1 references the exception description.

Operand 2 specifies the new attribute values. Operand 2 may be either a character constant or a space pointer to the modification template. When operand 3 is a constant, operand 2 is a character constant; when operand 3 is not a constant, operand 2 is a space pointer.

The value of operand 3 specifies the *modification option*. If the *modification option* is hex 01 and operand 2 specifies a space pointer, the format of the *modifying attributes* pointed to by operand 2 is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Template size	Char(8)	
0	0		Number of bytes provided for materialization (must be at least 10)	Bin(4)
4	4		Number of bytes available for materialization	Bin(4) +
8	8	Control flags	Char(2)	
8	8		Exception handling action	Bits 0-2

Offset Dec	Offset Hex	Field Name	Data Type and Length		
			000 =	Do not handle. (Ignore occurrence of exception and continue processing.)	
			001 =	Do not handle. (Disable this exception description and continue to search this invocation for another exception description to handle the exception.)	
			010 =	Do not handle. (Continue to search for an exception description by resignaling the exception to the preceding invocation.)	
			100 =	Defer handling. (Save exception data for later exception handling.)	
			101 =	Pass control to the specified exception handler.	
8	8		No data		Bit 3
			0 =	Exception data is returned	
			1 =	Exception data is not returned	
8	8		Reserved (binary 0)		Bits 4-15
10	A	—- End —-			

If the exception description was in the deferred state prior to the modification, the deferred signal, if present, is lost.

When the **no data** field is set to *exception data is not returned*, no data is returned for the Retrieve Exception Data (RETEXCPD) or Test Exception (TESTEXCP) instructions, and the *number of bytes available* for materialization field is set to 0. This option can also be selected in the object definition table entry of the exception description.

If the *modification option* of operand 3 is a constant value of hex 01, then operand 2 may specify a character constant. The operand 2 constant has the same format as the control flags entry previously described.

If the *modification option* is hex 02, then operand 2 must specify a space pointer. The format of the modification is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size	Char(8)	
0	0		Number of bytes provided (must be at least 10 plus the length of the *compare value* in the exception description)	Bin(4)
4	4		Number of bytes available for materialization	Bin(4) +
8	8	Compare value length (maximum of 32 bytes)	Bin(2) +	
10	A	Compare value	Char(32)	
42	2A	—- End —-		

Note: Fields shown here with a plus sign (+) are ignored by the instruction.

The number of bytes in the *compare value* is dictated by the **compare value length** specified in the exception description as originally specified in the object definition table.

An external exception handling program can be modified by resolving addressability to a new program into the system pointer designated for the exception description.

The presence of user data is not a modifiable attribute of exception descriptions. If the exception description has user data, it can be modified by changing the value of the data object specified in the exception description.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3802 Template Size Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Modify Independent Index (MODINX)

Op Code (Hex)	Operand 1	Operand 2
0452	Independent index	Modification option

Operand 1: System pointer.

Operand 2: Character(4) scalar.

Bound program access

```
Built-in number for MODINX is 39.
MODINX (
        independent_index     : address of system pointer
        modification_option   : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: Modify the selected attributes of the *independent index* specified by operand 1 to have the values specified in operand 2. The *modification option*s specified in operand 2 have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Modification selection	Char(1)	
0	0		Reserved (binary 0)	Bit 0
0	0		Immediate update	Bit 1
			0 = Do not change immediate update attribute	
			1 = Change immediate update attribute	
0	0		Index coherency tracking	Bit 2
			0 = Do not change index coherency tracking attribute	
			1 = Change index coherency tracking attribute	
0	0		Reserved (binary 0)	Bits 3-7
1	1	New attribute value	Char(1)	
1	1		Reserved (binary 0)	Bit 0
1	1		Immediate update	Bit 1
			0 = No immediate update	
			1 = Immediate update	
1	1		Index coherency tracking	Bit 2
			0 = Do not track index coherency	
			1 = Track index coherency	
1	1		Reserved (binary 0)	Bits 3-7
2	2	Reserved (binary 0)	Char(2)	
4	4	—- End —-		

If the **modification selection immediate update** is binary 0, then the *immediate update* attribute is not changed. If the modification selection *immediate update* bit is binary 1, the *immediate update* attribute is changed to the **new attribute value immediate update** value.

If the *immediate update* attribute of the index was previously set to *no immediate update*, and it is being modified to *immediate update*, then the index is ensured before the attribute is modified.

If the **modification selection index coherency tracking** is binary 0, then the *index coherency tracking* attribute is not changed. If the modification selection *index coherency tracking* bit is binary 1, the *index coherency tracking* attribute is changed to the **new attribute value index coherency tracking** value.

Warning: Temporary Level 3 Header

Authorization Required

•

- Object management
  –
  – Operand 1
- Execute

–
– Contexts referenced for address resolution

Lock Enforcement

•
• Modify
  –
  – Operand 1
• Materialization
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C0E IASP Resources Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Modify Invocation Authority Attributes (MODINVAU)

Op Code (Hex)	**Operand 1**
0141	Modification template

Operand 1: Space pointer.

Bound program access
Built-in number for MODINVAU is 477. MODINVAU (modification_template : address)

Description: This instruction modifies the authority attributes of the invocation that issues the instruction. The authority attribute to be modified and its new value are indicated by operand 1.

The format of the *modification template* is described below.

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Option	Char(1)
		Hex 00 = Do not suppress adopted user profile authority	
		Hex 01 = Suppress adopted user profile authority	
1	1	—- End —-	

The *do not suppress adopted user profile authority* option allows normal propagation of program adopted user profile authority to subsequent invocations within the same thread.

The *suppress adopted user profile authority* option prevents any subsequent invocation from benefitting from the propagation of currently adopted user profile authority. Thus any authority adopted and propagated by earlier invocations or the current invocation cannot be used as a source of authority by subsequent invocations within the same thread. Any subsequent invocation to the invocation that used the *suppress adopted user profile authority* option of the MODINVAU instruction may adopt and propagate its owning user profile authority.

Once the invocation that issued the *suppress adopted user profile authority* option returns to its caller, then adopted user profile authority propagated by previous invocations can be used by subsequent invocations.

The *suppress adopted user profile authority* option and the suppress adopted user profile authority *yes* option of the CALLX or XCTL instructions provide the same function for programs.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

22 Object Access

2202 Object Destroyed

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Modify Space Attributes (MODS)

Op Code (Hex)	Operand 1	Operand 2
0062	System object	Size or space modification template

Operand 1: System pointer.

Operand 2: Binary scalar or character(28) scalar.

Bound program access

```
Built-in number for MODS1 is 28.
MODS1 (
       system_object  : address of system pointer
       size           : address of signed binary(4)  OR
                        address of unsigned binary(4)
)

  -- OR --

Built-in number for MODS2 is 29.
MODS2 (
       system_object                : address of system pointer
       space_modification_template  : address
)
```

Description: The attributes of the space associated with the *system object* specified for operand 1 are modified with the attribute values specified in operand 2.

At any security level, if the thread execution state is user state and the object addressed by operand 1 has secondary associated spaces, an attempt to truncate or delete any of the associated spaces of the addressed object will result in an *invalid space modification* (hex 3602) exception.

If the thread execution state is user state and the machine security level attribute has a value of hex 40 or greater then

-
- If operand 1 addresses a program object, the associated spaces of the program object can not be modified and an *invalid space modification* (hex 3602) exception is signaled.
- If the hardware storage protection of the object addressed by operand 1 is not read/write from user state, the associated spaces of the addressed object can not be modified and a *space extension/truncation* (hex 3601) exception is signaled.

The operand 2 *space modification template* is specified with one of two formats. The abbreviated format, operand 2 specified as a binary scalar, only provides for modifying the *size of space* attribute. The full format, operand 2 specified as a character scalar, provides for modifying the full set of space attributes.

When operand 2 is a binary value, it specifies the size in bytes to which the space size is to be modified. The current allocation of the space is extended or truncated accordingly to match as closely as possible the specified size. The modified space size will be of at least the size specified. The actual size allocated is dependent upon the algorithm used within the specific implementation of the machine.

When operand 2 is a character scalar, it specifies a selection of space attribute values to be used to modify the attributes of the space. Associated spaces can be modified, created or destroyed by this instruction. Not all attributes can be modified for existing associated spaces, so some template fields apply only when an associated space is being created, i.e. when a primary associated space of fixed length size zero is modified or when a secondary associated space is created. More detail is provided below within descriptions of individual fields.

The operand 2 character scalar must be at least 28 bytes long and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Modification selection	Char(4)	
0	0		Modify space length attribute **0 =** No **1 =** Yes	Bit 0
0	0		Modify size of space **0 =** No **1 =** Yes	Bit 1
0	0		Modify initial value of space **0 =** No **1 =** Yes	Bit 2
0	0		Modify space alignment **0 =** No **1 =** Yes This field must have a value of 0 if an existing space is being modified. Otherwise an *invalid space modification* (hex 3602) exception is signaled.	Bit 3
0	0		Modify initialize space attribute **0 =** No **1 =** Yes	Bit 4
0	0		Reinitialize space **0 =** No **1 =** Yes	Bit 5
0	0		Modify automatically extend space attribute **0 =** No **1 =** Yes	Bit 6
0	0		Create secondary associated space **0 =** No **1 =** Yes If the thread execution state is user state, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled. This restriction applies at all system security levels.	Bit 7
0	0		Reserved (binary 0)	Bit 8
0	0		Modify hardware storage protection enforcement	Bit 9

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = No	
			1 = Yes	
			If the thread execution state is user state, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled. This restriction applies at all system security levels.	
			If the object is not an independent index, process control space, or a space, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled.	
0	0		Modify expanded transfer size advisory	Bit 10
			0 = No	
			1 = Yes	
0	0		Modify spreading the space object	Bit 11
			0 = No	
			1 = Yes	
0	0		Reserved (binary 0)	Bits 12-31
4	4	Indicator attributes	Char(4)	
4	4		Reserved (binary 0)	Bit 0
4	4		Space length	Bit 1
			0 = Fixed length	
			1 = Variable length	
4	4		Initialize space	Bit 2
			0 = Initialize	
			1 = Do not initialize	
4	4		Automatically extend space	Bit 3
			0 = No	
			1 = Yes	
4	4		Reserved (binary 0)	Bits 4-14
4	4		Hardware storage protection level	Bits 15-16

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			00 = Reference and modify allowed for user state programs	
			01 = Only reference allowed for user state programs	
			10 = Invalid (undefined)	
			11 = No reference or modify allowed for user state programs	
4	4		Reserved (binary 0)	Bits 17-20
4	4		Always enforce hardware storage protection of this space	Bit 21
			0 = Enforce hardware storage protection of this space only when hardware storage protection is being enforced for all storage.	
			1 = Enforce hardware storage protection of this space at all times.	
4	4		Reserved (binary 0)	Bits 22-31
8	8	Maximum size of secondary associated space This field is ignored when *create secondary associated space* is 0.	Bin(4)	
12	C	Size of space	Bin(4) or UBin(4)	
16	10	Initial value of space	Char(1)	
17	11	Performance class	Char(4)	
17	11		Space alignment	Bit 0
			0 = The space associated with the object is modified to allow proper alignment of pointers at 16-byte alignments within the space.	
			1 = The space associated with the object is modified to allow proper alignment of input/output buffers at 512-byte alignments within the space. Note that this also allows proper 16-byte alignment of pointers.	
			The value of this field is ignored when the *machine chooses space alignment* field has a value of 1.	
17	11		Reserved There is no modification selection option for the field in CRTS that corresponds to this bit. So, currently this bit is ignored.	Bit 1
17	11		Spread the space object	Bit 2

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = All extensions to the space object should be on one storage device, if possible.	
			1 = All extensions to the space object should be spread across multiple storage devices, if possible.	
17	11		Machine chooses space alignment	Bit 3
			0 = The space alignment indicated by the *space alignment* field is performed.	
			1 = The machine will choose the space alignment most beneficial to performance, which may reduce maximum space capacity. When the *modify space alignment* field has a value of 1 and this value is specified, the *space alignment* field is ignored, but the alignment chosen will be a multiple of 512.	
17	11		Reserved	Bits 4-23
			There are no modification selection options for the fields in CRTS that correspond to these bits. So, currently these bits are ignored.	
20	14		Expanded transfer size advisory	Char(1)
21	15	Reserved (binary 0)	Char(1)	
22	16	Secondary associated space number If the thread execution state is user state, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled. This restriction applies at all system security levels.	UBin(2)	
24	18	Reserved (binary 0)	Char(4)	
28	1C	—- End —-		

The **modification selection** indicator fields select the modifications to be performed on the space.

The **modify space length attribute** modification selection field controls whether or not the space length attribute is to be modified. When *yes* is specified, the value of the **space length** indicator is used to modify the space to the specified *fixed* or *variable* length attribute. When *no* is specified, the *space length* indicator attribute value is ignored and the *space length* attribute is not modified.

The *modify space length attribute* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify size of space** modification selection field controls whether or not the allocation size of the space is to be modified. When *yes* is specified, the current allocation of the space is extended or truncated accordingly to match as closely as possible the specified size in the **size of space** field. The modified size will be at least the size specified. The actual size allocated is dependent upon the algorithm used within the specific implementation of the machine. When *no* is specified, the current allocation of the space is not modified and the *size of space* field is ignored.

Modification of the *size of space attribute* for a space of fixed length can only be performed in conjunction with modification of the *space length* attribute. In this case, the *space length* attribute may be modified to

the same fixed length attribute or to the variable length attribute. An attempt to modify the *size of space* attribute for a space of fixed length without modification of the *space length* attribute results in the signaling of the *space extension/truncation* (hex 3601) exception. Modification of the *size of space* attribute for a space of variable length can always be performed separately from a modification of the *space length* attribute.

When the *size of space* attribute is to be modified, if the value of the *size of space* field is negative or specifies a size larger than that for the largest space that can be associated with the object, the *space extension/truncation* (hex 3601) exception is signaled.

The *modify size of space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify initial value of space** modification selection field controls whether or not the *initial value of space* attribute is to be modified. When *yes* is specified, the value of the **initial value of space** field is used to modify the corresponding attribute of this space. This byte value will be used to initialize any new space allocations for this space due to an extension to the size of space attribute on the current execution of this instruction as well as any subsequent modifications. When *no* is specified, the *initial value of space* field is ignored and the *initial value of space* attribute is not modified.

The *modify initial value of space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify space alignment** modification selection field controls whether or not the *space alignment* and *machine chooses space alignment* attributes of the specified system object are to be modified. When *yes* is specified, the values of the *space alignment* and *machine chooses space alignment* fields are used to modify the space alignment of the specified system object. When *no* is specified, the space alignment attributes of the specified system object are not modified.

The *modify space alignment* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify initialize space attribute** modification selection field controls whether or not the *initialize space attribute* is to be modified. When *yes* is specified, the value of the **initialize space** indicator attribute is used to modify that attribute of the specified space to the specified value. When *no* is specified, the *initialize space* indicator attribute value is ignored and the *initialize space* attribute is not modified.

Changing the value of the *initialize space* attribute only affects whether or not future extensions of the space will be initialized or not. That is, it is the state of this attribute at the time of allocation of the storage for a space that determines whether that newly allocated storage area will be initialized to the initial value specified for the space. Modifications of this attribute subsequent to the allocation of storage to a space have no effect on the value of that previously allocated storage area.

The *modify initialize space attribute* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **reinitialize space** modification selection field controls whether the storage allocated to the space is to be reinitialized in its entirety. When *no* is specified, the space is not reinitialized. When *yes* is specified, the space is reinitialized. This re-initialization is performed after all other attribute modifications which may also have been specified on the instruction have been made. Thus changes to the *size of the space*, the *initial value of the space*, etc. will be put into effect and be considered the current attributes of the space for purposes of the re-initialization. The byte value used for the re-initialization is the current initial value for the space.

Note that specifying *yes* for the *reinitialize space* modification selection field for a space with current attributes of fixed length size zero results in no operation, because such a space has no allocated storage to reinitialize. Also, note that re-initialization of a space will have the side effect of resetting partial damage for a space object containing the space if the space object had previously been marked as having partial damage. This only applies to space objects; i.e. re-initialization of an associated space does not have the side effect of resetting partial damage for the MI object containing it.

The *reinitialize space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify automatically extend space** attribute modification selection field controls whether or not the *automatically extend space* attribute is to be modified. When *yes* is specified, the value of the **automatically extend space** indicator attribute is used to modify that attribute of the specified space to the specified value. When *no* is specified, the *automatically extend space* indicator attribute value is ignored and the *automatically extend space* attribute is not modified. The *automatically extend space* attribute can only be specified as *yes* when the *space length* attribute for the space is already *variable* length, or when the *space length* attribute is being modified to *variable* length. Invalid specification of the *automatically extend space* attribute results in the signaling of the *invalid space modification* (hex 3602) exception.

The *modify automatically extend space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **create secondary associated space** field indicates if a secondary associated space is to be created for the object. When this field is set to *yes*, most of the operand 2 template fields are used. However, all other *modification selection* fields are ignored, as are the *spread the space*, and *always enforce hardware storage protection of this space* fields. All secondary associated spaces are protected with hardware storage protection at all times.

The secondary associated space to be created is indicated by the *secondary associated space number* field. If the specified space already exists, or the object already has its maximum number of secondary associated spaces, the *invalid space modification* (hex 3602) exception is signalled.

The **modify hardware storage protection enforcement** selection field controls whether the *enforce hardware storage protection of this space at all times* attribute is to be modified. When *yes* is specified, the value of the *enforce hardware storage protection of this space at all times* field is used to control when hardware storage protection will be enforced for the primary associated space of a process control space, independent index, or space object that is being modified by this operation. When this attribute is selected and an existing space is being modified, the existing hardware storage protection level in effect for the space will be unchanged, but will either be enforced at all times, or only when hardware storage protection is enforced for all storage.

The **modify expanded transfer size advisory** selection field controls whether the *expanded transfer size advisory* attribute is to be modified. When *yes* is specified, the value of **expanded transfer size advisory** specifies the desired number of pages to be transferred between main store and auxiliary storage for implicit access state changes. This value is only an advisory; the machine may use a value of its choice for performing access state changes under some circumstances. For example, the machine may limit the transfer size to a smaller value than is specified. A value of zero is an explicit indication that the machine should use the machine default storage transfer size for this object.

Modification of the *expanded transfer size advisory* is only supported for space objects. Attempts to modify associated spaces of other system objects will cause the *invalid space modification* (hex 3602) exception to be signalled.

The **modify spreading the space object** attribute modification selection field controls whether or not the *spread the space object* attribute is to be modified. When *yes* is specified and *spread the space object* is binary

1, extensions to the space object will be spread across multiple storage devices, if possible. When *yes* is specified and *spread the space object* is zero, extensions to the space object will be contained on one storage device, if possible. When *no* is specified, the *spread the space object* field is ignored and the current attribute setting for the space object is unchanged. The actual storage devices used are dependent upon the algorithm used within the specific implementation of the machine. Only the new allocations of the space object are affected; the existing portion of the space object is not modified.

The *modify spreading the space object* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception. This field is ignored for a system object that is not a space object.

The **hardware storage protection level** field determines the type(s) of accesses that are allowed to the space. This field is only used when creating associated spaces. That is, it is only used when extending the size of a primary associated space of fixed length and size equal to zero, or creating secondary associated space(s). For all other modifications it is ignored.

Modification to or from the state of a space being fixed length of size zero can not be performed for the following objects:

Byte stream file

Cursor

Data space

Directory

Program (when attempted while in user state on a security level 40 or higher system).

Space

Modification to or from the state of a space being fixed length of size zero might not be permitted for the following objects if they were created with an internal format incompatible with this change, which could have occurred for these objects if they were created before V4R4:

Class of service description

Controller description

Logical unit description

Mode description

Network description

If such a modification is attempted for the objects listed above, under the circumstances described above, the *invalid space modification* (hex 3602) exception is signaled.

Specifying the **largest size of space needed** value allows the machine, under certain circumstances, to select usage of an internal storage allocation unit which best utilizes the internal addressing resources within the machine. Note that the internal storage allocation unit selected can alter the maximum modification size of the associated space for the object. However, the machine will always use an internal storage allocation unit that will allow for extension of the space to at least the value specified in the largest size of space needed field. The maximum size to which the space can be modified is dependent upon specific implementations of the machine and can vary with different machine implementations.

The **secondary associated space number** field is used to indicate which secondary space is to be created or modified. When this field is zero, the primary associated space of the space object is modified. If this field is not zero and no secondary associated spaces are allowed for the object, the *scalar value invalid* (hex 3203) exception will be signalled.

A fixed length space of size zero is defined by the machine to have no internal storage allocation. Due to this, a modification to or from this state is, in essence, the same as a destroy or create for the space associated with the specified system object. The effect of modifying to this state is similar to destroying the associated space in that address references to the space through previously set pointers will result in signaling of the *object destroyed* (hex 2202) exception. When a primary associated space is destroyed by using this method, any secondary associated spaces for the object are also destroyed. To the contrary, modifying the space attributes from this state is similar to creating an associated space in that the Set Space Pointer from Pointer (SETSPPFP) instruction can be used to set a space pointer to the start of storage within the associated space and the allocated space storage can be used to contain space data.

The extension and truncation of a space is always by an implementation-defined multiple of 256 bytes. This means that if, for example, the implementation defined multiple is 2 (or 512 bytes), any modification of the space size will be in increments of 512 bytes.

Warning: Temporary Level 3 Header

Authorization Required

•

- Object management
  –
  – Operand 1
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Contexts referenced for address resolution
- Object control
  –
  – Operand 1 (when operand 2 is binary)
- Modify
  –
  – Operand 1 (when operand 2 is character)

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

0A Authorization

- 0A01 Unauthorized for Operation

10 Damage Encountered

- 1004 System Object Damage State
- 1005 Authority Verification Terminated Due to Damaged Object
- 1044 Partial System Object Damage

1A Lock State

- 1A01 Invalid Lock State

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded
- 1C04 Object Storage Limit Exceeded
- 1C0E IASP Resources Exceeded
- 1C11 Independent ASP Varied Off

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2207 Authority Verification Terminated Due to Destroyed Object
- 2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

3602 Invalid Space Modification

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Modify Space Attributes (MODS)

Op Code (Hex)	Operand 1	Operand 2
0062	System object	Size or space modification template

Operand 1: System pointer.

Operand 2: Binary scalar or character(28) scalar.

Bound program access

```
Built-in number for MODS1 is 28.
MODS1 (
       system_object  : address of system pointer
       size           : address of signed binary(4)  OR
                        address of unsigned binary(4)
)

  -- OR --

Built-in number for MODS2 is 29.
MODS2 (
       system_object                : address of system pointer
       space_modification_template  : address
)
```

Description: The attributes of the space associated with the *system object* specified for operand 1 are modified with the attribute values specified in operand 2.

At any security level, if the thread execution state is user state and the object addressed by operand 1 has secondary associated spaces, an attempt to truncate or delete any of the associated spaces of the addressed object will result in an *invalid space modification* (hex 3602) exception.

If the thread execution state is user state and the machine security level attribute has a value of hex 40 or greater then

•

- If operand 1 addresses a program object, the associated spaces of the program object can not be modified and an *invalid space modification* (hex 3602) exception is signaled.
- If the hardware storage protection of the object addressed by operand 1 is not read/write from user state, the associated spaces of the addressed object can not be modified and a *space extension/truncation* (hex 3601) exception is signaled.

The operand 2 *space modification template* is specified with one of two formats. The abbreviated format, operand 2 specified as a binary scalar, only provides for modifying the *size of space* attribute. The full format, operand 2 specified as a character scalar, provides for modifying the full set of space attributes.

When operand 2 is a binary value, it specifies the size in bytes to which the space size is to be modified. The current allocation of the space is extended or truncated accordingly to match as closely as possible the specified size. The modified space size will be of at least the size specified. The actual size allocated is dependent upon the algorithm used within the specific implementation of the machine.

When operand 2 is a character scalar, it specifies a selection of space attribute values to be used to modify the attributes of the space. Associated spaces can be modified, created or destroyed by this instruction. Not all attributes can be modified for existing associated spaces, so some template fields apply only when an associated space is being created, i.e. when a primary associated space of fixed length size zero is modified or when a secondary associated space is created. More detail is provided below within descriptions of individual fields.

The operand 2 character scalar must be at least 28 bytes long and have the following format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Modification selection	Char(4)	
0	0		Modify space length attribute	Bit 0
			0 = No	
			1 = Yes	
0	0		Modify size of space	Bit 1
			0 = No	
			1 = Yes	
0	0		Modify initial value of space	Bit 2
			0 = No	
			1 = Yes	
0	0		Modify space alignment	Bit 3

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = No **1 =** Yes This field must have a value of 0 if an existing space is being modified. Otherwise an *invalid space modification* (hex 3602) exception is signaled.	
0	0		Modify initialize space attribute **0 =** No **1 =** Yes	Bit 4
0	0		Reinitialize space **0 =** No **1 =** Yes	Bit 5
0	0		Modify automatically extend space attribute **0 =** No **1 =** Yes	Bit 6
0	0		Create secondary associated space **0 =** No **1 =** Yes If the thread execution state is user state, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled. This restriction applies at all system security levels.	Bit 7
0	0		Reserved (binary 0)	Bit 8
0	0		Modify hardware storage protection enforcement **0 =** No **1 =** Yes If the thread execution state is user state, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled. This restriction applies at all system security levels. If the object is not an independent index, process control space, or a space, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled.	Bit 9
0	0		Modify expanded transfer size advisory **0 =** No **1 =** Yes	Bit 10

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0		Modify spreading the space object	Bit 11
			0 = No	
			1 = Yes	
0	0		Reserved (binary 0)	Bits 12-31
4	4	Indicator attributes	Char(4)	
4	4		Reserved (binary 0)	Bit 0
4	4		Space length	Bit 1
			0 = Fixed length	
			1 = Variable length	
4	4		Initialize space	Bit 2
			0 = Initialize	
			1 = Do not initialize	
4	4		Automatically extend space	Bit 3
			0 = No	
			1 = Yes	
4	4		Reserved (binary 0)	Bits 4-14
4	4		Hardware storage protection level	Bits 15-16
			00 = Reference and modify allowed for user state programs	
			01 = Only reference allowed for user state programs	
			10 = Invalid (undefined)	
			11 = No reference or modify allowed for user state programs	
4	4		Reserved (binary 0)	Bits 17-20
4	4		Always enforce hardware storage protection of this space	Bit 21
			0 = Enforce hardware storage protection of this space only when hardware storage protection is being enforced for all storage.	
			1 = Enforce hardware storage protection of this space at all times.	
4	4		Reserved (binary 0)	Bits 22-31

Offset Dec	Hex	Field Name	Data Type and Length
8	8	Maximum size of secondary associated space This field is ignored when *create secondary associated space* is 0.	Bin(4)
12	C	Size of space	Bin(4) or UBin(4)
16	10	Initial value of space	Char(1)
17	11	Performance class	Char(4)
17	11		Space alignment (Bit 0) **0 =** The space associated with the object is modified to allow proper alignment of pointers at 16-byte alignments within the space. **1 =** The space associated with the object is modified to allow proper alignment of input/output buffers at 512-byte alignments within the space. Note that this also allows proper 16-byte alignment of pointers. The value of this field is ignored when the *machine chooses space alignment* field has a value of 1.
17	11		Reserved (Bit 1) There is no modification selection option for the field in CRTS that corresponds to this bit. So, currently this bit is ignored.
17	11		Spread the space object (Bit 2) **0 =** All extensions to the space object should be on one storage device, if possible. **1 =** All extensions to the space object should be spread across multiple storage devices, if possible.
17	11		Machine chooses space alignment (Bit 3) **0 =** The space alignment indicated by the *space alignment* field is performed. **1 =** The machine will choose the space alignment most beneficial to performance, which may reduce maximum space capacity. When the *modify space alignment* field has a value of 1 and this value is specified, the *space alignment* field is ignored, but the alignment chosen will be a multiple of 512.
17	11		Reserved (Bits 4-23) There are no modification selection options for the fields in CRTS that correspond to these bits. So, currently these bits are ignored.
20	14		Expanded transfer size advisory Char(1)
21	15	Reserved (binary 0)	Char(1)
22	16	Secondary associated space number	UBin(2)

Offset Dec	Hex	Field Name	Data Type and Length
		If the thread execution state is user state, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled. This restriction applies at all system security levels.	
24	18	Reserved (binary 0)	Char(4)
28	1C	—- End —-	

The **modification selection** indicator fields select the modifications to be performed on the space.

The **modify space length attribute** modification selection field controls whether or not the space length attribute is to be modified. When *yes* is specified, the value of the **space length** indicator is used to modify the space to the specified *fixed* or *variable* length attribute. When *no* is specified, the *space length* indicator attribute value is ignored and the *space length* attribute is not modified.

The *modify space length attribute* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify size of space** modification selection field controls whether or not the allocation size of the space is to be modified. When *yes* is specified, the current allocation of the space is extended or truncated accordingly to match as closely as possible the specified size in the **size of space** field. The modified size will be at least the size specified. The actual size allocated is dependent upon the algorithm used within the specific implementation of the machine. When *no* is specified, the current allocation of the space is not modified and the *size of space* field is ignored.

Modification of the *size of space attribute* for a space of fixed length can only be performed in conjunction with modification of the *space length* attribute. In this case, the *space length* attribute may be modified to the same fixed length attribute or to the variable length attribute. An attempt to modify the *size of space* attribute for a space of fixed length without modification of the *space length* attribute results in the signaling of the *space extension/truncation* (hex 3601) exception. Modification of the *size of space* attribute for a space of variable length can always be performed separately from a modification of the *space length* attribute.

When the *size of space* attribute is to be modified, if the value of the *size of space* field is negative or specifies a size larger than that for the largest space that can be associated with the object, the *space extension/truncation* (hex 3601) exception is signaled.

The *modify size of space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify initial value of space** modification selection field controls whether or not the *initial value of space* attribute is to be modified. When *yes* is specified, the value of the **initial value of space** field is used to modify the corresponding attribute of this space. This byte value will be used to initialize any new space allocations for this space due to an extension to the size of space attribute on the current execution of this instruction as well as any subsequent modifications. When *no* is specified, the *initial value of space* field is ignored and the *initial value of space* attribute is not modified.

The *modify initial value of space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify space alignment** modification selection field controls whether or not the *space alignment* and *machine chooses space alignment* attributes of the specified system object are to be modified. When *yes* is

specified, the values of the *space alignment* and *machine chooses space alignment* fields are used to modify the space alignment of the specified system object. When *no* is specified, the space alignment attributes of the specified system object are not modified.

The *modify space alignment* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify initialize space attribute** modification selection field controls whether or not the *initialize space attribute* is to be modified. When *yes* is specified, the value of the **initialize space** indicator attribute is used to modify that attribute of the specified space to the specified value. When *no* is specified, the *initialize space* indicator attribute value is ignored and the *initialize space* attribute is not modified.

Changing the value of the *initialize space* attribute only affects whether or not future extensions of the space will be initialized or not. That is, it is the state of this attribute at the time of allocation of the storage for a space that determines whether that newly allocated storage area will be initialized to the initial value specified for the space. Modifications of this attribute subsequent to the allocation of storage to a space have no effect on the value of that previously allocated storage area.

The *modify initialize space attribute* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **reinitialize space** modification selection field controls whether the storage allocated to the space is to be reinitialized in its entirety. When *no* is specified, the space is not reinitialized. When *yes* is specified, the space is reinitialized. This re-initialization is performed after all other attribute modifications which may also have been specified on the instruction have been made. Thus changes to the *size of the space*, the *initial value of the space*, etc. will be put into effect and be considered the current attributes of the space for purposes of the re-initialization. The byte value used for the re-initialization is the current initial value for the space.

Note that specifying *yes* for the *reinitialize space* modification selection field for a space with current attributes of fixed length size zero results in no operation, because such a space has no allocated storage to reinitialize. Also, note that re-initialization of a space will have the side effect of resetting partial damage for a space object containing the space if the space object had previously been marked as having partial damage. This only applies to space objects; i.e. re-initialization of an associated space does not have the side effect of resetting partial damage for the MI object containing it.

The *reinitialize space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify automatically extend space** attribute modification selection field controls whether or not the *automatically extend space* attribute is to be modified. When *yes* is specified, the value of the **automatically extend space** indicator attribute is used to modify that attribute of the specified space to the specified value. When *no* is specified, the *automatically extend space* indicator attribute value is ignored and the *automatically extend space* attribute is not modified. The *automatically extend space* attribute can only be specified as *yes* when the *space length* attribute for the space is already *variable* length, or when the *space length* attribute is being modified to *variable* length. Invalid specification of the *automatically extend space* attribute results in the signaling of the *invalid space modification* (hex 3602) exception.

The *modify automatically extend space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **create secondary associated space** field indicates if a secondary associated space is to be created for the object. When this field is set to *yes*, most of the operand 2 template fields are used. However, all other

modification selection fields are ignored, as are the *spread the space*, and *always enforce hardware storage protection of this space* fields. All secondary associated spaces are protected with hardware storage protection at all times.

The secondary associated space to be created is indicated by the *secondary associated space number* field. If the specified space already exists, or the object already has its maximum number of secondary associated spaces, the *invalid space modification* (hex 3602) exception is signalled.

The **modify hardware storage protection enforcement** selection field controls whether the *enforce hardware storage protection of this space at all times* attribute is to be modified. When *yes* is specified, the value of the *enforce hardware storage protection of this space at all times* field is used to control when hardware storage protection will be enforced for the primary associated space of a process control space, independent index, or space object that is being modified by this operation. When this attribute is selected and an existing space is being modified, the existing hardware storage protection level in effect for the space will be unchanged, but will either be enforced at all times, or only when hardware storage protection is enforced for all storage.

The **modify expanded transfer size advisory** selection field controls whether the *expanded transfer size advisory* attribute is to be modified. When *yes* is specified, the value of **expanded transfer size advisory** specifies the desired number of pages to be transferred between main store and auxiliary storage for implicit access state changes. This value is only an advisory; the machine may use a value of its choice for performing access state changes under some circumstances. For example, the machine may limit the transfer size to a smaller value than is specified. A value of zero is an explicit indication that the machine should use the machine default storage transfer size for this object.

Modification of the *expanded transfer size advisory* is only supported for space objects. Attempts to modify associated spaces of other system objects will cause the *invalid space modification* (hex 3602) exception to be signalled.

The **modify spreading the space object** attribute modification selection field controls whether or not the *spread the space object* attribute is to be modified. When *yes* is specified and *spread the space object* is binary 1, extensions to the space object will be spread across multiple storage devices, if possible. When *yes* is specified and *spread the space object* is zero, extensions to the space object will be contained on one storage device, if possible. When *no* is specified, the *spread the space object* field is ignored and the current attribute setting for the space object is unchanged. The actual storage devices used are dependent upon the algorithm used within the specific implementation of the machine. Only the new allocations of the space object are affected; the existing portion of the space object is not modified.

The *modify spreading the space object* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception. This field is ignored for a system object that is not a space object.

The **hardware storage protection level** field determines the type(s) of accesses that are allowed to the space. This field is only used when creating associated spaces. That is, it is only used when extending the size of a primary associated space of fixed length and size equal to zero, or creating secondary associated space(s). For all other modifications it is ignored.

Modification to or from the state of a space being fixed length of size zero can not be performed for the following objects:

Byte stream file

Cursor

Data space

Directory

Program (when attempted while in user state on a security level 40 or higher system).

Space

Modification to or from the state of a space being fixed length of size zero might not be permitted for the following objects if they were created with an internal format incompatible with this change, which could have occurred for these objects if they were created before V4R4:

Class of service description

Controller description

Logical unit description

Mode description

Network description

If such a modification is attempted for the objects listed above, under the circumstances described above, the *invalid space modification* (hex 3602) exception is signaled.

Specifying the **largest size of space needed** value allows the machine, under certain circumstances, to select usage of an internal storage allocation unit which best utilizes the internal addressing resources within the machine. Note that the internal storage allocation unit selected can alter the maximum modification size of the associated space for the object. However, the machine will always use an internal storage allocation unit that will allow for extension of the space to at least the value specified in the largest size of space needed field. The maximum size to which the space can be modified is dependent upon specific implementations of the machine and can vary with different machine implementations.

The **secondary associated space number** field is used to indicate which secondary space is to be created or modified. When this field is zero, the primary associated space of the space object is modified. If this field is not zero and no secondary associated spaces are allowed for the object, the *scalar value invalid* (hex 3203) exception will be signalled.

A fixed length space of size zero is defined by the machine to have no internal storage allocation. Due to this, a modification to or from this state is, in essence, the same as a destroy or create for the space associated with the specified system object. The effect of modifying to this state is similar to destroying the associated space in that address references to the space through previously set pointers will result in signaling of the *object destroyed* (hex 2202) exception. When a primary associated space is destroyed by using this method, any secondary associated spaces for the object are also destroyed. To the contrary, modifying the space attributes from this state is similar to creating an associated space in that the Set

Space Pointer from Pointer (SETSPPFP) instruction can be used to set a space pointer to the start of storage within the associated space and the allocated space storage can be used to contain space data.

The extension and truncation of a space is always by an implementation-defined multiple of 256 bytes. This means that if, for example, the implementation defined multiple is 2 (or 512 bytes), any modification of the space size will be in increments of 512 bytes.

Warning: Temporary Level 3 Header

Authorization Required

-
- Object management
  -
  - Operand 1
- Execute
  -
  - Contexts referenced for address resolution

Lock Enforcement

-
- Materialize
  -
  - Contexts referenced for address resolution
- Object control
  -
  - Operand 1 (when operand 2 is binary)
- Modify
  -
  - Operand 1 (when operand 2 is character)

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C0E IASP Resources Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

3602 Invalid Space Modification

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Modify Space Attributes (MODS)

Op Code (Hex)	Operand 1	Operand 2
0062	System object	Size or space modification template

Operand 1: System pointer.

Operand 2: Binary scalar or character(28) scalar.

Bound program access

```
Built-in number for MODS1 is 28.
MODS1 (
       system_object  : address of system pointer
       size           : address of signed binary(4)  OR
                        address of unsigned binary(4)
)

  -- OR --

Built-in number for MODS2 is 29.
MODS2 (
       system_object                 : address of system pointer
       space_modification_template   : address
)
```

Description: The attributes of the space associated with the *system object* specified for operand 1 are modified with the attribute values specified in operand 2.

At any security level, if the thread execution state is user state and the object addressed by operand 1 has secondary associated spaces, an attempt to truncate or delete any of the associated spaces of the addressed object will result in an *invalid space modification* (hex 3602) exception.

If the thread execution state is user state and the machine security level attribute has a value of hex 40 or greater then

•

- If operand 1 addresses a program object, the associated spaces of the program object can not be modified and an *invalid space modification* (hex 3602) exception is signaled.
- If the hardware storage protection of the object addressed by operand 1 is not read/write from user state, the associated spaces of the addressed object can not be modified and a *space extension/truncation* (hex 3601) exception is signaled.

The operand 2 *space modification template* is specified with one of two formats. The abbreviated format, operand 2 specified as a binary scalar, only provides for modifying the *size of space* attribute. The full format, operand 2 specified as a character scalar, provides for modifying the full set of space attributes.

When operand 2 is a binary value, it specifies the size in bytes to which the space size is to be modified. The current allocation of the space is extended or truncated accordingly to match as closely as possible the specified size. The modified space size will be of at least the size specified. The actual size allocated is dependent upon the algorithm used within the specific implementation of the machine.

When operand 2 is a character scalar, it specifies a selection of space attribute values to be used to modify the attributes of the space. Associated spaces can be modified, created or destroyed by this instruction. Not all attributes can be modified for existing associated spaces, so some template fields apply only when an associated space is being created, i.e. when a primary associated space of fixed length size zero is modified or when a secondary associated space is created. More detail is provided below within descriptions of individual fields.

The operand 2 character scalar must be at least 28 bytes long and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Modification selection	Char(4)	
0	0		Modify space length attribute	Bit 0
			0 = No	
			1 = Yes	
0	0		Modify size of space	Bit 1
			0 = No	
			1 = Yes	
0	0		Modify initial value of space	Bit 2
			0 = No	
			1 = Yes	
0	0		Modify space alignment	Bit 3
			0 = No	
			1 = Yes This field must have a value of 0 if an existing space is being modified. Otherwise an *invalid space modification* (hex 3602) exception is signaled.	
0	0		Modify initialize space attribute	Bit 4
			0 = No	
			1 = Yes	
0	0		Reinitialize space	Bit 5
			0 = No	
			1 = Yes	
0	0		Modify automatically extend space attribute	Bit 6
			0 = No	
			1 = Yes	
0	0		Create secondary associated space	Bit 7

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = No **1 =** Yes If the thread execution state is user state, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled. This restriction applies at all system security levels.	
0	0		Reserved (binary 0)	Bit 8
0	0		Modify hardware storage protection enforcement **0 =** No **1 =** Yes If the thread execution state is user state, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled. This restriction applies at all system security levels. If the object is not an independent index, process control space, or a space, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled.	Bit 9
0	0		Modify expanded transfer size advisory **0 =** No **1 =** Yes	Bit 10
0	0		Modify spreading the space object **0 =** No **1 =** Yes	Bit 11
0	0		Reserved (binary 0)	Bits 12-31
4	4	Indicator attributes	Char(4)	
4	4		Reserved (binary 0)	Bit 0
4	4		Space length **0 =** Fixed length **1 =** Variable length	Bit 1
4	4		Initialize space **0 =** Initialize **1 =** Do not initialize	Bit 2
4	4		Automatically extend space	Bit 3

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = No	
			1 = Yes	
4	4		Reserved (binary 0)	Bits 4-14
4	4		Hardware storage protection level	Bits 15-16
			00 = Reference and modify allowed for user state programs	
			01 = Only reference allowed for user state programs	
			10 = Invalid (undefined)	
			11 = No reference or modify allowed for user state programs	
4	4		Reserved (binary 0)	Bits 17-20
4	4		Always enforce hardware storage protection of this space	Bit 21
			0 = Enforce hardware storage protection of this space only when hardware storage protection is being enforced for all storage.	
			1 = Enforce hardware storage protection of this space at all times.	
4	4		Reserved (binary 0)	Bits 22-31
8	8	Maximum size of secondary associated space This field is ignored when *create secondary associated space* is 0.	Bin(4)	
12	C	Size of space	Bin(4) or UBin(4)	
16	10	Initial value of space	Char(1)	
17	11	Performance class	Char(4)	
17	11		Space alignment	Bit 0
			0 = The space associated with the object is modified to allow proper alignment of pointers at 16-byte alignments within the space.	
			1 = The space associated with the object is modified to allow proper alignment of input/output buffers at 512-byte alignments within the space. Note that this also allows proper 16-byte alignment of pointers.	
			The value of this field is ignored when the *machine chooses space alignment* field has a value of 1.	
17	11		Reserved	Bit 1

Offset Dec	Hex	Field Name	Data Type and Length	
			There is no modification selection option for the field in CRTS that corresponds to this bit. So, currently this bit is ignored.	
17	11		Spread the space object	Bit 2
			0 = All extensions to the space object should be on one storage device, if possible.	
			1 = All extensions to the space object should be spread across multiple storage devices, if possible.	
17	11		Machine chooses space alignment	Bit 3
			0 = The space alignment indicated by the *space alignment* field is performed.	
			1 = The machine will choose the space alignment most beneficial to performance, which may reduce maximum space capacity. When the *modify space alignment* field has a value of 1 and this value is specified, the *space alignment* field is ignored, but the alignment chosen will be a multiple of 512.	
17	11		Reserved	Bits 4-23
			There are no modification selection options for the fields in CRTS that correspond to these bits. So, currently these bits are ignored.	
20	14		Expanded transfer size advisory	Char(1)
21	15	Reserved (binary 0)	Char(1)	
22	16	Secondary associated space number	UBin(2)	
		If the thread execution state is user state, this field must be 0. Otherwise an *invalid space modification* (hex 3602) exception is signaled. This restriction applies at all system security levels.		
24	18	Reserved (binary 0)	Char(4)	
28	1C	—- End —-		

The **modification selection** indicator fields select the modifications to be performed on the space.

The **modify space length attribute** modification selection field controls whether or not the space length attribute is to be modified. When *yes* is specified, the value of the **space length** indicator is used to modify the space to the specified *fixed* or *variable* length attribute. When *no* is specified, the *space length* indicator attribute value is ignored and the *space length* attribute is not modified.

The *modify space length attribute* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify size of space** modification selection field controls whether or not the allocation size of the space is to be modified. When *yes* is specified, the current allocation of the space is extended or truncated accordingly to match as closely as possible the specified size in the **size of space** field. The modified size

will be at least the size specified. The actual size allocated is dependent upon the algorithm used within the specific implementation of the machine. When *no* is specified, the current allocation of the space is not modified and the *size of space* field is ignored.

Modification of the *size of space attribute* for a space of fixed length can only be performed in conjunction with modification of the *space length* attribute. In this case, the *space length* attribute may be modified to the same fixed length attribute or to the variable length attribute. An attempt to modify the *size of space* attribute for a space of fixed length without modification of the *space length* attribute results in the signaling of the *space extension/truncation* (hex 3601) exception. Modification of the *size of space* attribute for a space of variable length can always be performed separately from a modification of the *space length* attribute.

When the *size of space* attribute is to be modified, if the value of the *size of space* field is negative or specifies a size larger than that for the largest space that can be associated with the object, the *space extension/truncation* (hex 3601) exception is signaled.

The *modify size of space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify initial value of space** modification selection field controls whether or not the *initial value of space* attribute is to be modified. When *yes* is specified, the value of the **initial value of space** field is used to modify the corresponding attribute of this space. This byte value will be used to initialize any new space allocations for this space due to an extension to the size of space attribute on the current execution of this instruction as well as any subsequent modifications. When *no* is specified, the *initial value of space* field is ignored and the *initial value of space* attribute is not modified.

The *modify initial value of space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify space alignment** modification selection field controls whether or not the *space alignment* and *machine chooses space alignment* attributes of the specified system object are to be modified. When *yes* is specified, the values of the *space alignment* and *machine chooses space alignment* fields are used to modify the space alignment of the specified system object. When *no* is specified, the space alignment attributes of the specified system object are not modified.

The *modify space alignment* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify initialize space attribute** modification selection field controls whether or not the *initialize space attribute* is to be modified. When *yes* is specified, the value of the **initialize space** indicator attribute is used to modify that attribute of the specified space to the specified value. When *no* is specified, the *initialize space* indicator attribute value is ignored and the *initialize space* attribute is not modified.

Changing the value of the *initialize space* attribute only affects whether or not future extensions of the space will be initialized or not. That is, it is the state of this attribute at the time of allocation of the storage for a space that determines whether that newly allocated storage area will be initialized to the initial value specified for the space. Modifications of this attribute subsequent to the allocation of storage to a space have no effect on the value of that previously allocated storage area.

The *modify initialize space attribute* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **reinitialize space** modification selection field controls whether the storage allocated to the space is to be reinitialized in its entirety. When *no* is specified, the space is not reinitialized. When *yes* is specified, the space is reinitialized. This re-initialization is performed after all other attribute modifications which

may also have been specified on the instruction have been made. Thus changes to the *size of the space*, the *initial value of the space*, etc. will be put into effect and be considered the current attributes of the space for purposes of the re-initialization. The byte value used for the re-initialization is the current initial value for the space.

Note that specifying *yes* for the *reinitialize space* modification selection field for a space with current attributes of fixed length size zero results in no operation, because such a space has no allocated storage to reinitialize. Also, note that re-initialization of a space will have the side effect of resetting partial damage for a space object containing the space if the space object had previously been marked as having partial damage. This only applies to space objects; i.e. re-initialization of an associated space does not have the side effect of resetting partial damage for the MI object containing it.

The *reinitialize space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **modify automatically extend space** attribute modification selection field controls whether or not the *automatically extend space* attribute is to be modified. When *yes* is specified, the value of the **automatically extend space** indicator attribute is used to modify that attribute of the specified space to the specified value. When *no* is specified, the *automatically extend space* indicator attribute value is ignored and the *automatically extend space* attribute is not modified. The *automatically extend space* attribute can only be specified as *yes* when the *space length* attribute for the space is already *variable* length, or when the *space length* attribute is being modified to *variable* length. Invalid specification of the *automatically extend space* attribute results in the signaling of the *invalid space modification* (hex 3602) exception.

The *modify automatically extend space* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception.

The **create secondary associated space** field indicates if a secondary associated space is to be created for the object. When this field is set to *yes*, most of the operand 2 template fields are used. However, all other *modification selection* fields are ignored, as are the *spread the space*, and *always enforce hardware storage protection of this space* fields. All secondary associated spaces are protected with hardware storage protection at all times.

The secondary associated space to be created is indicated by the *secondary associated space number* field. If the specified space already exists, or the object already has its maximum number of secondary associated spaces, the *invalid space modification* (hex 3602) exception is signalled.

The **modify hardware storage protection enforcement** selection field controls whether the *enforce hardware storage protection of this space at all times* attribute is to be modified. When *yes* is specified, the value of the *enforce hardware storage protection of this space at all times* field is used to control when hardware storage protection will be enforced for the primary associated space of a process control space, independent index, or space object that is being modified by this operation. When this attribute is selected and an existing space is being modified, the existing hardware storage protection level in effect for the space will be unchanged, but will either be enforced at all times, or only when hardware storage protection is enforced for all storage.

The **modify expanded transfer size advisory** selection field controls whether the *expanded transfer size advisory* attribute is to be modified. When *yes* is specified, the value of **expanded transfer size advisory** specifies the desired number of pages to be transferred between main store and auxiliary storage for implicit access state changes. This value is only an advisory; the machine may use a value of its choice for performing access state changes under some circumstances. For example, the machine may limit the transfer size to a smaller value than is specified. A value of zero is an explicit indication that the machine should use the machine default storage transfer size for this object.

Modification of the *expanded transfer size advisory* is only supported for space objects. Attempts to modify associated spaces of other system objects will cause the *invalid space modification* (hex 3602) exception to be signalled.

The **modify spreading the space object** attribute modification selection field controls whether or not the *spread the space object* attribute is to be modified. When *yes* is specified and *spread the space object* is binary 1, extensions to the space object will be spread across multiple storage devices, if possible. When *yes* is specified and *spread the space object* is zero, extensions to the space object will be contained on one storage device, if possible. When *no* is specified, the *spread the space object* field is ignored and the current attribute setting for the space object is unchanged. The actual storage devices used are dependent upon the algorithm used within the specific implementation of the machine. Only the new allocations of the space object are affected; the existing portion of the space object is not modified.

The *modify spreading the space object* modification selection field may not be set to *yes* for a packed secondary associated space. An attempt to do so will result in an *invalid space modification* (hex 3602) exception. This field is ignored for a system object that is not a space object.

The **hardware storage protection level** field determines the type(s) of accesses that are allowed to the space. This field is only used when creating associated spaces. That is, it is only used when extending the size of a primary associated space of fixed length and size equal to zero, or creating secondary associated space(s). For all other modifications it is ignored.

Modification to or from the state of a space being fixed length of size zero can not be performed for the following objects:

Byte stream file

Cursor

Data space

Directory

Program (when attempted while in user state on a security level 40 or higher system).

Space

Modification to or from the state of a space being fixed length of size zero might not be permitted for the following objects if they were created with an internal format incompatible with this change, which could have occurred for these objects if they were created before V4R4:

Class of service description

Controller description

Logical unit description

Mode description

Network description

If such a modification is attempted for the objects listed above, under the circumstances described above, the *invalid space modification* (hex 3602) exception is signaled.

Specifying the **largest size of space needed** value allows the machine, under certain circumstances, to select usage of an internal storage allocation unit which best utilizes the internal addressing resources within the machine. Note that the internal storage allocation unit selected can alter the maximum modification size of the associated space for the object. However, the machine will always use an internal storage allocation unit that will allow for extension of the space to at least the value specified in the largest size of space needed field. The maximum size to which the space can be modified is dependent upon specific implementations of the machine and can vary with different machine implementations.

The **secondary associated space number** field is used to indicate which secondary space is to be created or modified. When this field is zero, the primary associated space of the space object is modified. If this field is not zero and no secondary associated spaces are allowed for the object, the *scalar value invalid* (hex 3203) exception will be signalled.

A fixed length space of size zero is defined by the machine to have no internal storage allocation. Due to this, a modification to or from this state is, in essence, the same as a destroy or create for the space associated with the specified system object. The effect of modifying to this state is similar to destroying the associated space in that address references to the space through previously set pointers will result in signaling of the *object destroyed* (hex 2202) exception. When a primary associated space is destroyed by using this method, any secondary associated spaces for the object are also destroyed. To the contrary, modifying the space attributes from this state is similar to creating an associated space in that the Set Space Pointer from Pointer (SETSPPFP) instruction can be used to set a space pointer to the start of storage within the associated space and the allocated space storage can be used to contain space data.

The extension and truncation of a space is always by an implementation-defined multiple of 256 bytes. This means that if, for example, the implementation defined multiple is 2 (or 512 bytes), any modification of the space size will be in increments of 512 bytes.

Warning: Temporary Level 3 Header

Authorization Required

•

- Object management
  –
  – Operand 1
- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Contexts referenced for address resolution
- Object control

  –
  – Operand 1 (when operand 2 is binary)
- Modify
  –
  – Operand 1 (when operand 2 is character)

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C0E IASP Resources Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

3602 Invalid Space Modification

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Multiply (MULT)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
MULT 104B		Product	Multiplicand	Multiplier	
MULTR 124B		Product	Multiplicand	Multiplier	
MULTI 184B	Indicator options	Product	Multiplicand	Multiplier	Indicator targets
MULTIR 1A4B	Indicator options	Product	Multiplicand	Multiplier	Indicator targets
MULTB 1C4B	Branch options	Product	Multiplicand	Multiplier	Branch targets
MULTBR 1E4B	Branch options	Product	Multiplicand	Multiplier	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4-7:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
MULTS 104B		Product/Multiplicand	Multiplier	
MULTSR 134B		Product/Multiplicand	Multiplier	
MULTIS 194B	Indicator options	Product/Multiplicand	Multiplier	Indicator targets
MULTISR 1B4B	Indicator options	Product/Multiplicand	Multiplier	Indicator targets
MULTBS 1D4B	Branch options	Product/Multiplicand	Multiplier	Branch targets
MULTBSR 1F4B	Branch options	Product/Multiplicand	Multiplier	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The *product* is the result of multiplying the *multiplicand* and the *multiplier.*

Operands can have floating-point, packed or zoned decimal, signed or unsigned binary type.

Source operands are the *multiplicand* and *multiplier.* The receiver operand is the *product.*

If operands are not of the same type, source operands are converted according to the following rules:

1. If any one of the operands has floating point type, source operands are converted to floating point type.
2. Otherwise, if any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.
3. Otherwise, the binary operands are converted as follows.
   a. If an unsigned binary(2) source operand is used with a signed binary operand of any length, the unsigned binary(2) is viewed as a signed binary(4).
   b. If both source operands are signed binary (including cases resulting from use of 3a (page 1003)), then a signed operation, of the length of the longer operand, is done.
   c. If both source operands are unsigned binary(2), then an unsigned 2-byte operation is done.
   d. If either source operand is unsigned binary(4), then an unsigned 4-byte operation is done with overflow detection disabled until the assignment to the receiver.

Source operands are multiplied according to their type. Floating point operands are multiplied using floating point multiplication. Packed decimal operands are multiplied using packed decimal multiplication. Unsigned binary multiplication is used with unsigned source operands, except as noted above. Signed binary operands are multiplied using two's complement binary multiplication.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary multiplication execute faster than either packed decimal or floating point multiplication.

The operands must be numeric with any implicit conversions occurring according to the rules of arithmetic operations as outlined in the Arithmetic Operations.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

If the *multiplicand* operand or the *multiplier* operand has a value of 0, the result of the multiplication is a zero product.

For a decimal operation, no alignment of the assumed decimal point is performed for the *multiplier* and *multiplicand* operands.

The operation occurs using the specified lengths of the *multiplicand* and *multiplier* operands with no logical zero padding on the left necessary.

Floating-point multiplication uses exponent addition and significand multiplication.

For nonfloating-point computations and for significand multiplication for floating-point operations, the multiplication operation is performed according to the rules of algebra. Unsigned binary operands are treated as positive numbers for the algebra.

The result of the operation is copied into the *product* operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the *product* operand, aligned at the assumed decimal point of the *product* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations outlined in the Arithmetic Operations.

For the optional round form of the instruction, specification of a floating-point receiver operand is invalid.

For fixed-point operations in programs that request to be notified of size exceptions, if nonzero digits are truncated from the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For floating-point operations involving a fixed-point receiver field (if nonzero digits would be truncated from the left end of the resultant value), an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point product operand, if the exponent of the resultant value is either too large or too small to be represented in the *product* field, the *floating-point overflow* (hex 0C06) exception or the *floating-point underflow* (hex 0C07) exception is signaled.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Conditions:

•

- Positive-The algebraic value of the numeric scalar *product* is positive.
- Negative-The algebraic value of the numeric scalar *product* is negative.
- Zero-The algebraic value of the numeric scalar *product* is zero.
- Unordered-The value assigned a floating-point *product* operand is NaN.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Negate (NEG)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
NEG 1056		Receiver	Source	
NEGI 1856	Indicator options	Receiver	Source	Indicator targets
NEGB 1C56	Branch options	Receiver	Source	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

•
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand [2-5]
NEGS 1156		Receiver/Source	
NEGIS 1956	Indicator options	Receiver/Source	Indicator targets
NEGBS 1D56	Branch options	Receiver/Source	Branch targets

Operand 1: Numeric variable scalar.

Operand 2-5:

•
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The numeric value in the *source* operand is changed as if it had been multiplied by a negative one (-1). The result is placed in the *receiver* operand.

The sign changing of the *source* operand value (positive to negative and negative to positive) is performed as follows:

•
- Binary
  –
  – Extract the numeric value and form the twos complement of it.
- Packed/Zoned
  –
  – Extract the numeric value and force its sign to positive if it is negative or to negative if it is positive.
- Floating-point
  –
  – Extract the numeric value and force the significand sign to positive if it is negative or to negative if it is positive.

The result of the operation is copied into the *receiver* operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the receiver operand, aligned at the assumed decimal point of the *receiver* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations outlined in the Arithmetic Operations. If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled. An attempt to negate a maximum negative signed binary value to a signed binary scalar of the same size also results in a *size* (hex 0C0A) exception. If a packed or zoned 0 is negated, the result is always positive 0.

When the *source* floating-point operand represents not-a-number, the sign field of the *source* is not forced to positive and this value is not altered in the *receiver*.

For a fixed-point operation, if significant digits are truncated from the left end of the resultant value, a *size* (hex 0C0A) exception is signaled. An attempt to negate a maximum negative binary value into a binary scalar of the same size also results in a *size* (hex 0C0A) exception.

For floating-point operations that involve a fixed-point *receiver*, if nonzero digits would be truncated from the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point *receiver* operand, if the exponent of the resultant value is either too large or too small to be represented in the *receiver*, the *floating-point overflow* (hex 0C06) exception and the *floating-point underflow* (hex 0C07) exception are signaled.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled or if the size exception was suppressed, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Conditions:

•

- Positive-The algebraic value of the *receiver* operand is positive.
- Negative-The algebraic value of the *receiver* operand is negative.
- Zero-The algebraic value of the *receiver* operand is zero.
- Unordered-The value assigned a floating-point *receiver* operand is NaN.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

No Operation (NOOP)

Op Code (Hex)
0000

Description: No function is performed. The instruction consists of an operation code and no operands. The instruction may not be branched to and is not counted as an instruction in the instruction stream. The instruction may be used for inserting gaps in the instruction stream. These gaps allow instructions with adjacent instruction addresses to be physically separated.

The instruction may precede or follow any machine instruction except the End instruction, and any number of No Operation instructions may exist in succession.

No Operation and Skip (NOOPS)

Op Code (Hex)	Operand 1
0001	Skip count

Operand 1: Unsigned immediate value.

Description: This instruction performs no function other than to indicate a specific number of bytes within the instruction stream that are to be skipped during encapsulation. It consists of an operation code and 1 operand. Operand 1 is an unsigned immediate value that contains the number of bytes between this instruction and the next instruction to be processed. These bytes are skipped during the encapsulation of this program. A value of zero for operand 1 indicates that no bytes are to be skipped between this instruction and the next instruction to be processed.

If the operand 1 skip count indicates that the next instruction to process is beyond the end of the instruction stream, an *invalid operand value range* (hex 2A08) exception is signaled.

This instruction may be used to insert gaps in the instruction stream in such a manner that allows instructions with adjacent instruction addresses to not be physically adjacent.

This instruction may not be branched to, and is not counted as an instruction in the instruction stream.

The instruction may precede or follow any machine instruction except the End instruction, and any number of No Operation and Skip instructions may exist in succession.

Note:

When this instruction is used in an existing program template, the following items within the template may be adversely affected:

-
- The actual count of instructions may be altered to be different than the count of instructions that is specified in the program template header.
- Object definitions that reference instructions may now be out of range or may not reference the intended instruction.

The actual number of bytes skipped includes the bytes containing the instruction plus the number of bytes specified by the skip count value. The number of bytes skipped per template version is as follows:

•
- Version 0 = 4 plus the skip count value.
- Version 1 = 5 plus the skip count value.

Not (NOT)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-4]
NOT 108A		Receiver	Source	
NOTI 188A	Indicator options	Receiver	Source	Indicator targets
NOTB 1C8A	Branch options	Receiver	Source	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character variable scalar or numeric variable scalar.

Operand 3-4:

•
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand [2-3]
NOTS 118A		Receiver/Source	
NOTIS 198A	Indicator options	Receiver/Source	Indicator targets
NOTBS 1D8A	Branch options	Receiver/Source	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2-3:

•
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Boolean **not** operation is performed on the string value in the *source* operand. The resulting string is placed in the *receiver* operand.

The operands may be character or numeric scalars. They are both interpreted as bit strings. Substringing is supported for both character and numeric operands.

The length of the operation is equal to the length of the *source* operand.

The bit values of the result are determined as follows:

Source Bit	Result Bit
0	1
1	0

The result value is then placed (left-adjusted) in the *receiver* operand with truncating or padding taking place on the right. The pad value used in this instruction is a hex 00 byte.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the *source* operand is that the result is all zero and the instruction's resultant condition is zero. When a null substring reference is specified for the *receiver*, a result is not set and the instruction's resultant condition is zero regardless of the value of the *source* operand.

When the *receiver* operand is a numeric variable scalar, it is possible that the result produced will not be a valid value for the numeric type. This can occur due to padding with hex 00, due to truncation, or due to the resultant bit string produced by the instruction. The instruction completes normally and signals no exceptions for these conditions.

Resultant Conditions:

•

- Zero-The bit value for the bits of the scalar *receiver* operand is either all zero or a null substring reference is specified for the *receiver*.
- Not zero-The bit value for the bits of the scalar *receiver* operand is not all zero.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

NPM Procedure Parameter List Address (NPM_PARMLIST_ADDR)

Bound program access
Built-in number for NPMPARMLISTADDR is 143. NPM_PARMLIST_ADDR ( ) : space pointer(16) that points to the New Program Model parameter list

Description: The address of the New Program Model parameter list received by the current invocation is returned.

This function cannot be used by procedures defined to be a program entry procedure. Otherwise, an *instruction stream not valid* (hex 2A1B) exception will be signalled during module creation.

Bound Procedure Parameter List Format: Calls to procedures use the parameter list format shown below. The parameter list is required to be quadword aligned.

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Operational descriptor list address	Char(16)
16	10	Machine work area	Char(16)
32	20	Arguments	Char(*)
*	*	—- End —-	

The **operational descriptor list address** is used to pass the address of an operational descriptor to the current invocation. If an operational descriptor is provided by the caller, the machine will store the address in this field as a space pointer. If an operational descriptor is not provided, the contents of this field are undefined. (That is, the machines does not set this field to a special value to indicate the absence of an operational descriptor list address.) The Machine Interface does not define the layout of the operational descriptor list, and it is the MI program's responsibility to pass an operational descriptor list to a procedure which expects it.

The **machine work area** is reserved for use by the machine.

Arguments is a variable length field used to pass argument values to the current invocation. Even though the actual mechanism used by the machine to pass argument values is not visible at the Machine Interface, this field is large enough to accommodate all passed argument values, with each argument being aligned on its natural boundary. At most 400 arguments may be passed.

The rules for natural alignment are shown in Table 1 (page 1014). For packed decimal arguments, the length in bytes is derived from the number of digits, `numberofDigits`, using the following formula: `((numberofDigits/2) + 1)`. For zoned decimal separate leading sign and zoned decimal separate trailing sign arguments, the byte length is one greater than the length in digits.

Table 1. Rules for natural alignment

Length in Bytes	Alignment
length = 1	byte
length = 2	halfword
length = 3, 4	word
length = 5, 6, 7, 8	doubleword
other lengths	quadword

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

OPM Parameter Address (OPM_PARM_ADDR)

Bound program access

```
Built-in number for OPMPARMADDR is 7.
OPM_PARM_ADDR (
               parameter_number   : unsigned binary(4) value which specifies
                                    a non-bound program parameter list
                                    entry
) :   space pointer(16) to the parameter list entry specified by
      parameter_number
```

Description: The entry in the non-bound program parameter list specified by *parameter number* is returned.

If the parameter number referenced is greater than the number of parameters passed to the program entry procedure, an address is returned which will cause a *parameter reference violation* (hex 0801) exception to be signaled when it is used.

This function can only be used by procedures defined to be a program entry procedure. Otherwise, an *instruction stream not valid* (hex 2A1B) exception will be signaled during module creation.

Notes:

1. The non-bound program operand passing protocol is "call by reference", and hence the OPM_PARM_ADDR built-in is architected to be returning the reference addresses that are passed.
2. Faster code will be generated when the parameter number passed to the built-in is a literal. Still faster code will be generated when the parameter number passed to the built-in is a literal which is less than or equal to the *minimum number of parameters required* value associated with this program entry procedure.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

OPM Parameter Count (OPM_PARM_CNT)

Bound program access

```
Built-in number for OPMPARMCNT is 8.
OPM_PARM_CNT (
) :   unsigned binary(4) value which specifies the number of non-bound
      program parameters passed to the program entry procedure
```

Description: The parameter count in the non-bound program parameter list received by the program entry procedure is returned.

This function can only be used by procedures defined to be a program entry procedure. Otherwise, an *instruction stream not valid* (hex 2A1B) exception will be signaled during module creation.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

Or (OR)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
OR 1097		Receiver	Source 1	Source 2	
ORI 1897	Indicator options	Receiver	Source 1	Source 2	Indicator targets
ORB 1C97	Branch options	Receiver	Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character scalar or numeric scalar.

Operand 4-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-4]
ORS 1197		Receiver/Source 1	Source 2	
ORIS 1997	Indicator options	Receiver/Source 1	Source 2	Indicator targets
ORBS 1D97	Branch options	Receiver/Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3-4:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Boolean **or** operation is performed on the string values in the source operands. The resulting string is placed in the *receiver* operand.

The operands may be character or numeric scalars. They are both interpreted as bit strings. Substringing is supported for both character and numeric operands.

The length of the operation is equal to the length of the longer of the two source operands. The shorter of the two operands is logically padded on the right with hex 00. The excess bytes in the longer operand are assigned to the results.

The bit values of the result are determined as follows:

Source 1 Bit	Source 2 Bit	Result Bit
0	0	0
0	1	1
1	0	1
1	1	1

The result value is then placed (left-adjusted) in the *receiver* operand with truncating or padding taking place on the right. The pad value used in this instruction is a hex 00.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1, 2, and 3. The effect of specifying a null substring reference for one source operand is that the other source operand is **or**ed with an equal length string of all hex 00s. This causes the value of the other operand to be assigned to the result. When a null substring reference is specified for both source operands, the result is all zero and the instruction's resultant condition is zero. When a null substring reference is specified for the *receiver*, a result is not set and the instruction's resultant condition is zero regardless of the values of the source operands.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

When the *receiver* operand is a numeric variable scalar, it is possible that the result produced will not be a valid value for the numeric type. This can occur due to padding with hex 00, due to truncation, or due to the resultant bit string produced by the instruction. The instruction completes normally and signals no exceptions for these conditions.

Resultant Conditions:

-
- Zero-The bit value for the bits of the scalar *receiver* operand is either all zero or a null substring reference is specified for the *receiver*.
- Not zero-The bit value for the bits of the scalar *receiver* operand is not all zero.

Authorization Required

-
- None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

OR String (ORSTR)

Bound program access

```
Built-in number for ORSTR is 451.
ORSTR (
       receiver_string        : address of aggregate(*)
       first_source_string    : address of aggregate(*)
       second_source_string   : address of aggregate(*)
       string_length          : unsigned binary(4,8) value which specifies
                                the length of the three strings
)
```

Description: Each byte value of the *first source string*, for the number of bytes indicated by *string length*, is logically **or**ed with the corresponding byte value of the *second source string*, on a bit-by-bit basis. The results are placed in the *receiver string*. If the strings overlap in storage, predictable results occur only if the overlap is fully coincident.

If the space(s) indicated by the three addresses are not long enough to contain the number of bytes indicated by *string length*, a *space addressing violation* (hex 0601) is signalled. Partial results in this case are unpredictable.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Override Program Attributes (OVRPGATR)

Op Code (Hex)	Operand 1	Operand 2
0006	Attribute identification	Attribute modifier

Operand 1: Unsigned immediate value.

Operand 2: Unsigned immediate value.

Description: This program creation control instruction allows one of a set of program attributes specified below to be overridden. The overridden program attribute is in effect until it is changed by another OVRPGATR instruction. The initial program attributes are set to the ones specified when the program is created. These same initial program attributes are the ones that are materialized when a Materialize Program (MATPG) is done. That is, the OVRPGATR instruction has no effect on the materialized attributes.

The OVRPGATR instruction consists of an operation code and two operands. Operand 1 is an unsigned immediate value that contains a representation of which program attribute is to be overridden. Operand 2 is an unsigned immediate value that contains a representation of how the program attribute is to be overridden.

This instruction may not be branched to, and is not counted as an instruction in the instruction stream.

The instruction may precede or follow any machine instruction.

The program attributes defined by operand 1 is overridden according to the following selection values:

Attribute Identification	Attribute Description
1	Array constrainment attribute Allowed values for operand 2: **1 =** Constrain array references **2 =** Do not constrain array references **3 =** Fully unconstrain array references **4 =** Terminate override of array constrainment attributes and resume use of the attributes specified in the program template
2	String constrainment attribute Allowed values for operand 2: **1 =** Constrain string references **2 =** Do not constrain string references **3 =** Terminate override of string constrainment attribute and resume use of the attribute specified in the program template
3	Suppress binary size exception attribute Allowed values for operand 2: **1 =** Suppress binary size exceptions **2 =** Do not suppress binary size exceptions **3 =** Terminate override of suppression of binary size exception attribute and resume use of the attribute specified in the program template
4	Suppress decimal data exception attribute Allowed values for operand 2: **1 =** Suppress decimal data exceptions **2 =** Do not suppress decimal data exceptions **3 =** Terminate override of suppression of decimal data exception attribute and resume use of the attribute specified in the program template
5	Copy Bytes with Pointers (CPYBWP) alignment data check attribute Allowed values for operand 2: **1 =** Constrain CPYBWP to require like alignment of operands (default) **2 =** Do not constrain CPYBWP to require like alignment of operands
6	Compare Pointer for Space Addressibility (CMPPSPAD) null pointer tolerance attribute Allowed values for operand 2: **1 =** Signal pointer does not exist exceptions for operands 1 and 2 (default) **2 =** Do not signal pointer does not exist exceptions for operands 1 and 2

PCO Pointer (PCOPTR)

Bound program access

`Built-in number for PCOPTR is 144.`
`PCOPTR (`
`) :  space pointer(16) to the PCO associated with the activation`

Description: A space pointer is returned, which points to the first byte of the PCO (process communication object) for the process which owns the program activation associated with the current

invocation. If the current invocation is a client executing in a shared activation group then the pointer returned refers to an object owned by a process other than the current process.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

Propagate Byte (PROPB)

Bound program access

```
Built-in number for PROPB is 10.
PROPB (
       target_string       : address of aggregate(*)
       source_byte         : signed binary(4) - rightmost byte specifies
                                                the source byte value  OR
                             unsigned binary(1)  OR
                             aggregate(1)
       propagation_count   : unsigned binary value which specifies the
                             number of times the value specified by
                             source byte should be propagated
) :   space pointer(16) to the target string
```

Description: The byte value specified by *source byte* is copied into the storage specified by *target string*. *Propagation count* specifies the number of bytes of storage into which this value is stored. It is assumed sufficient storage exists at the location specified by *target string*.

If the value of *progagation count* is 0, no storage is altered.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Reallocate Activation Group-Based Heap Space Storage (REALCHSS)

Op Code (Hex)	Operand 1	Operand 2
03BA	Space allocation	Size of space reallocation

Operand 1: Space pointer.

Operand 2: Binary(4) scalar.

Bound program access

```
Built-in number for REALCHSS is 117.
REALCHSS (
          space_allocation             : address
          size_of_space_reallocation   : signed binary(4)
) :   space pointer(16) to space reallocation
```

space allocation and *size of space_reallocation* correspond to operands 1 and 2 on the REALCHSS operation; the return value corresponds to operand 1 after the function completes.

Note: The term ″heap space″ in this instruction refers to an ″activation group-based heap space″.

Description: A new heap space storage allocation of at least the size indicated by operand 2 is provided from the same heap space as the original allocation, which is indicated by operand 1. The operand 1 space pointer is set to address the first byte of the new allocation, which will begin on a boundary at least as great as the minimum boundary specified when the heap space was created.

Each allocation associated with a heap space provides a continuum of contiguously addressable bytes. Individual allocations within a heap space have no addressability affinity with each other.

The maximum single allocation allowed is determined by the maximum single allocation size specified when the heap space was created. The maximum single allocation possible is (16M - 1 page) bytes. To determine the current page size use the MATRMD instruction.

Storage that is reallocated maintains the same mark/release status as the original allocation. If the original allocation was marked, the new allocation carries the same mark and will be released by a Free Activation Group-Based Heap Space Storage from Mark (FREHSSMK) which specifies that mark identifier.

The original heap space storage allocation will be freed. Subsequent references to the original allocation will cause unpredictable results.

The contents of the original allocation are preserved in the following fashion:

•

- If the new allocation size is greater than the original allocation size, the entire contents of the original allocation will appear in the new allocation. The contents of the rest of the new allocation are unpredictable unless initialization of heap allocations was specified when the heap space was created.
- If the new allocation size is less than or equal to the original allocation size, the new allocation will contain at least as much of the original allocation contents as the new allocation size allows.
- If the minimum boundary alignment value for the heap space indicates at least a 16 byte boundary, valid pointers will be preserved.

REALCHSS will signal an *object domain or hardware storage protection violation* (hex 4401) exception if a program running user state attempts to reallocate heap space storage in a heap space with a *domain* of *system*.

Operand 2 is not modified by the instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

45 Heap Space

4502 Invalid Request

4503 Heap Space Full

4504 Invalid Size Request

4505 Heap Space Destroyed

4506 Invalid Heap Space Condition

Reinitialize Static Storage (RINZSTAT)

Op Code (Hex)	Operand 1
RINZSTAT2 02D1	Activation template
RINZSTAT 02C1	Activation template

Operand 1: Space pointer.

Bound program access

```
Built-in number for RINZSTAT2 is 664.
RINZSTAT2 (
           activation_template   : address
)

OR
Built-in number for RINZSTAT is 417.
RINZSTAT (
          activation_template   : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Note

It is recommended that you use the RINZSTAT2 instruction which supports 8-byte activation group marks. 4-byte marks can wrap and produce unexpected results.

Description: This instruction reinitializes the static storage for eligible previously activated bound programs, including bound service programs. To be eligible, the program must have the *allow static storage re-initialization* attribute set. The activation mark of any affected program activation is not changed.

Operand 1 must contain a space pointer to the *activation template*.

The *activation template* must be quadword aligned. The format of the structure is different for the RINZSTAT and the RINZSTAT2 instructions.

Format of activation template for RINZSTAT2 instruction:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Activation template	Char(24)	
0	0		System pointer to the program	System pointe
16	10		Activation group mark	UBin(8)
			For Non-Bound programs, the following datatype should be used:	
16	10		Activation group mark (Non-Bound program)	
24	18	—- End —-		

Format of activation template for RINZSTAT instruction:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Activation template	Char(20)	
0	0		System pointer to the program	System pointer
16	10		Activation group mark	UBin(4)
20	14	—- End —-		

If the **system pointer to the program** is a null pointer value, all eligible activations of bound programs or bound service programs within the activation group specified by the *activation group mark* will be interrogated.

If the thread is in user state at the time this instruction is invoked and the activation group, specified by the *activation group mark*, is a system state activation group an *activation group access violation* (hex 2C12) exception will be signalled. The user must have adequate authority to all bound programs with the *allow static storage re-initialization* attribute in the activation group or none of the programs will be reinitialized.

If the *system pointer to the program* is not a null pointer value, only the program activation in the activation group specified, provided there is adequate authority and the program has the *allow static storage re-initialization* attribute will have its static storage reinitialized. If the activation group specified by the *activation group mark* is not found, an *activation group not found* (hex 2C13) exception will be signalled. If the program activation can not be found in the activation group, an *invalid operation for program* (hex 2C15) exception will be signalled. If the program does not have the *allow static storage re-initialization* attribute set, an *invalid operation for program* (hex 2C15) exception will be signalled.

The **activation group mark** uniquely identifies an activation group within a process. A value of zero is interpreted to be a request to use the activation group of the current invocation.

Exported data to the activation group will not be changed.

Warning: Temporary Level 3 Header

Usage Notes

Static storage is initialized by the machine each time a program (bound program or bound service program) is activated or reinitialized via this instruction. Only those static storage locations specified by the high-level language (HLL) compiler are initialized by the machine. Other locations are uninitialized. The machine can only initialize static storage with values which can be specified as constants at

compile-time. Complex values which can only be evaluated at run-time cannot be initialized using this mechanism. For example, the current time of day or the ″construction″ of a C++ object.

Some HLL compilers (e.g. C++) make use of additional run-time static initialization (*s-init*) mechanisms to initialize variables with complex values. The *s-init* mechanism is triggered by execution of the program entry point (PEP) of a bound program and is supported by language run-time code. The RINZSTAT instruction **does not** cause this *s-init* code to be re-executed. The use of the RINZSTAT instruction on a program containing *s-init* items can lead to an inconsistent internal state of the program.

Authorization Required

•

- Execute
  –
  – Program referenced by operand 1
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2C Program Execution

2C12 Activation Group Access Violation

2C13 Activation Group Not Found

2C15 Invalid Operation for Program

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Remainder (REM)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
REM 1073		Remainder	Dividend	Divisor	
REMI 1873	Indicator options	Remainder	Dividend	Divisor	Indicator targets
REMB 1C73	Branch options	Remainder	Dividend	Divisor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-5]
REMS 1173		Remainder/Dividend	Divisor	
REMIS 1973	Indicator options	Remainder/Dividend	Divisor	Indicator targets
REMBS 1D73	Branch options	Remainder/Dividend	Divisor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-5:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The *remainder* is the result of dividing the *dividend* by the *divisor* and placing the remainder in operand 1.

Operands can have packed or zoned decimal, signed or unsigned binary type.

Source operands are the *dividend* and *divisor.* The receiver operand is the *remainder.*

If operands are not of the same type, source operands are converted according to the following rules:

1. If any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.
2. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Source operands are divided according to their type. Packed decimal operands are divided using packed decimal division. Unsigned binary division is used with unsigned source operands. Signed binary operands are divided using two's complement binary division.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary division execute faster than packed decimal division.

The operands must be numeric with any implicit conversions occurring according to the rules of arithmetic operations as outlined in the Arithmetic Operations.

Floating-point is not supported for this instruction.

If the *divisor* has a numeric value of 0, a *zero divide* (hex 0C0B) exception is signaled. If the *dividend* has a value of 0, the result of the division is a zero value remainder.

For a decimal operation, the internal quotient value produced by the divide operation is always calculated with a precision of zero fractional digit positions. If necessary, internal alignment of the assumed decimal point for the *dividend* and *divisor* operands is performed to insure the correct precision for the resultant quotient value. These internal alignments are not subject to detection of the decimal point alignment exception. An internal quotient and the corresponding remainder value will be calculated for any combination of decimal attributes which may be specified for the instruction's operands. However, as described below, the assignment of the remainder value is limited to that portion of the remainder value which fits in the *remainder* operand.

If the *dividend* is shorter than the *divisor,* it is logically adjusted to the length of the *divisor.*

The division operation is performed according to the rules of algebra. Unsigned binary is treated as a positive number for the algebra. Before the remainder is calculated, an intermediate quotient is calculated. The attributes of this quotient are derived from the attributes of the dividend and divisor operands as follows:

Dividend	Divisor	Intermediate Quotient
IM,SBIN	SBIN(2)	
or	or	
SBIN(2)	SBIN(2)	
IM,SBIN	SBIN(4)	
or		
SBIN(2)		
IM,SBIN	SBIN(Q2,Q2)	SBIN(Q2,0)
or		
UBIN(2)		
IM,SBIN	UBIN(2)	
or	or	
SBIN(4)	UBIN(4)	
UBIN(2)	IM,SBIN	SBIN(2)
or	or	
UBIN(4)	SBIN(4)	
UBIN(2)	UBIN(2)	
or	or	
UBIN(4)	UBIN(4)	
SBIN(4)	IM,SBIN	
	or	
	SBIN(2)	
SBIN(4)	DECIMAL(Q2,Q2)	SBIN(Q2,0)
or		
UBIN(4)		
DECIMAL(Q1,Q1)	UBIN(4)	SBIN(Q1,0)
	or	
	UBIN(2)	
DECIMAL(Q1,Q1)	SBIN(4)	SBIN(Q1,0)
	or	
	UBIN(4)	
DECIMAL(Q1,Q1)	DECIMAL(Q2,Q2)	DECIMAL(Q1+Q,0)

Where Q = Larger of Q1 or Q2

```
IM = IMMEDIATE
SIM = SIGNED IMMEDIATE
SBIN = SIGNED BINARY
UBIN = UNSIGNED BINARY
DECIMAL = PACKED OR ZONED
```

After the intermediate quotient numeric value has been determined, the numeric value of the *remainder* operand is calculated as follows:

```
Remainder = Dividend - (Quotient*Divisor)
```

When signed arithmetic is used, the sign of the remainder is the same as that of the dividend unless the *remainder* has a value of 0. When the *remainder* has a value of 0, the sign of the *remainder* is positive.

The resultant value of the calculation is copied into the *remainder* operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the *remainder* operand, aligned at the assumed decimal point of the *remainder* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations as outlined in Arithmetic Operations.

If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled for those programs that request to be notified of size exceptions.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled in programs that request to be notified of size exceptions, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Conditions:

•

- Positive-The algebraic value of the numeric scalar *remainder* is positive.
- Negative-The algebraic value of the numeric scalar *remainder* is negative.
- Zero-The algebraic value of the numeric scalar *remainder* is zero.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C0A Size

0C0B Zero Divide

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Remove Independent Index Entry (RMVINXEN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0484	Receiver	Index	Option list	Argument

Operand 1: Space pointer or null.

Operand 2: System pointer.

Operand 3: Space pointer.

Operand 4: Space pointer.

Bound program access

```
Built-in number for RMVINXEN is 40.
RMVINXEN (
          receiver      : address OR
                          null operand
          index         : address of system pointer
          option_list   : address
          argument      : address
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The index entries identified by operands 3 and 4 are removed from the independent index identified by operand 2 and optionally returned in the *receiver* specified by operand 1. The maximum length of an independent index entry is either 120 bytes or 2,000 bytes depending on how the *maximum entry length* attribute field was specified when the index was created. Note that all indexes created in Version 3 Release 6 or later have a maximum entry length of 2,000 bytes.

The *option list* (operand 3) and the *argument* (operand 4) have the same format and meaning as the option list and search argument for the Find Independent Index Entry (FNDINXEN) instruction. The **return count** designates the number of index entries that were removed from the index.

The arguments removed are returned in the receiver field if a space pointer is specified for operand 1. If operand 1 is null, the entries removed from the index are not returned. If neither space pointer nor null is specified for operand 1, the entries are returned in the same way that entries are returned for the Find Independent Index Entry instruction (FNDINXEN).

Every entry removed causes the *entries removed* count to be incremented by 1. The current value of this count is available through the Materialize Independent Index Attributes (MATINXAT) instruction. The *entries removed* field must be less than 4,096.

Warning: Temporary Level 3 Header

Authorization Required

•

• Delete

  –

  – Operand 2

• Execute

  –

  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize

–
– Contexts referenced for address resolution
- Modify
  –
  – Operand 2

Exceptions

02 Access Group

0201 Object Ineligible for Access Group

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C0E IASP Resources Exceeded

1C11 Independent ASP Varied Off

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Resolve Data Pointer (RSLVDP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0163	Pointer for addressability to data object	Data object identification	Program

Operand 1: Data pointer.

Operand 2: Character(32) scalar or null.

Operand 3: System pointer or null.

Bound program access
Built-in number for RSLVDP is 385. RSLVDP (pointer_for_addressability_to_data_object : address of data pointer data_object_identification : address OR null operand program : address of system pointer OR null operand)

Description: A data pointer with addressability to and the attributes of an external scalar data element is returned in the storage area identified by operand 1. The following describes the instruction's function when operand 2 is null:

•

- If operand 1 does not contain a data pointer, an exception is signaled.
- If the data pointer specified by operand 1 is not resolved and has an initial value declaration, the instruction resolves the data pointer to the external scalar that the initial value describes. The initial value defines the external scalar to be located and, optionally, defines the program in which it is to be located. If the program name is specified in the initial value, only that program's activation entry is searched for the external scalar. If no program is specified, programs associated with the activation entries in the current activation group in which the program is executing, are searched in reverse order of the activation entries, and operand 3 is ignored. The current activation group for non-bound programs is the default activation group whose state is the same as the state of the process at the time the instruction is run.
- If the data pointer is currently resolved and defines an existing scalar, the instruction causes no operation, and no exception is signaled.

The following describes the instruction's function when operand 2 is not null:

•

- A data pointer that is resolved to the external scalar identified by operand 2 is returned in operand 1. Operand 2 is a 32-byte value that provides the name of the external scalar to be located.
- Operand 3 specifies a system pointer that identifies the program whose activation is to be searched for the external scalar definition. If operand 3 is null, the instruction searches all activations in the activation group from which the instruction is executed, starting with the most recent activation and continuing to the oldest. The activation under which the instruction is issued also participates in the search. If operand 3 is not null, the instruction searches the activation of the program addressed by the system pointer.

Warning: Temporary Level 3 Header

Authorization Required

•

- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range
- 0604 External Data Object Not Found

08 Argument/Parameter

- 0801 Parameter Reference Violation

0A Authorization

- 0A01 Unauthorized for Operation

10 Damage Encountered

- 1004 System Object Damage State
- 1005 Authority Verification Terminated Due to Damaged Object
- 1044 Partial System Object Damage

1A Lock State

- 1A01 Invalid Lock State

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2207 Authority Verification Terminated Due to Destroyed Object
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2404 Pointer Not Resolved

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Resolve System Pointer (RSLVSP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0164	Pointer for addressability to object	Resolve options	Context through which object is to be located	Authority to be set [1] (page 1048)

Operand 1: System pointer.

Operand 2: Character(34, 128) scalar or null.

Operand 3: System pointer or null.

Operand 4: Character(2) scalar or null.

Bound program access

```
Built-in number for RSLVSP is 30.
RSLVSP (
        pointer_for_addressability_to_object           : address of system pointer
        resolve_options                                : address OR
                                                         null operand
        context_through_which_object_is_to_be_located  : address of system pointer OR
                                                         null operand
        authority_to_be_set                            : address OR
                                                         null operand
)
```

Description: This instruction locates an object identified by a symbolic address and stores the object's addressability and authority [1] (page 1048) in a system pointer. A resolved system pointer is returned in operand 1 with addressability to a system object and the requested authority currently available to the thread for the object.

Note: The ownership flag is never set in the system pointer.

Operand 2 specifies the symbolic identification of the object to be located. Operand 3 identifies the context to be searched in order to locate the object. Operand 4 identifies the authority states to be set in the pointer. First, the instruction locates an object based on operands 1, 2 and 3. Then, the instruction sets the appropriate authority states in the system pointer. An *object not found* (hex 2201) exception is signaled if the object is not found.

The object to be located is either addressed through a machine context or a context object.

The following object types can only be addressed through the system ASP machine context (i.e. they cannot be addressed through an independent ASP machine context or a context object):

Hex 08 = User profile

Hex 10 = Logical unit description

Hex 11 = Network description

Hex 12 = Controller description

Hex 14 = Class of service description

Hex 15 = Mode description

Hex 16 = Network interface description

Hex 17 = Connection list

Hex 1D = Auxiliary server

The following object types can only be addressed through the system ASP machine context or an independent ASP machine context (i.e. they cannot be addressed through a context object):

Hex 04 = Context

Other objects types are addressed through a context object, which may reside in the system ASP, basic ASP, or an ASP group.

No two context objects with the same name and subtype can exist within a given ASP group or within all basic ASPs and the system ASP (combined). Also, a context object in the system ASP or basic ASP cannot have the same name and subtype as a context in an ASP group. But context objects that are in different ASP groups may have the same name and subtype.

An ASP group is a set of independent ASPs that are configured such that they always vary on and off together.

The search for the object to be located proceeds as follows:

-
- If an object is a type that can only reside in the system ASP machine context, then only that machine context is searched.
- Otherwise, if the *search method* field in the extended template is used to limit the search to a context object (or machine context) in a specified independent ASP or ASP group, then only that context object (or machine context) is searched.

- Otherwise, the name space of the current thread is searched. A name space is a list that identifies which machine contexts RSLVSP uses to search for a context object. The name space includes the system ASP and basic ASPs and may include one ASP group.

  (Note that the *search method* field can be used to limit the search to just the ASP groups of the current thread's name space.)

When a name space is used to resolve to an object, the method used to search the name space depends on the type of object being resolved to:

-
- If the object to be resolved is a context object, then each machine context in the name space is searched. If the context object is found, this instruction will resolve to the context. If the context object is not found, an *object not found* (hex 2201) exception is signaled.
- If the object to be resolved can reside *in* a context object and a containing context is specified, then the search algorithm depends on whether or not the context object has an alias context. A containing context object is specified in operand 3 (see description of operand 3) or when operand 1 defines the name of a context object in an initial value declaration (see description of behavior when operand 2 is null).

  An alias context is a context object on an independent ASP which is searched before searching the specified context in the system ASP or basic ASP. Note that only context objects in the system ASP or basic ASPs can have an alias context. The alias context will always be in the ASP group of the current thread's name space.

  If the name space contains an ASP group and the ASP group has an alias context for the containing context object, then this instruction searches for the specified object in the alias context first. If the object is found, the instruction resolves to that object. If the object is not found (or if the context does not have an alias context) the instruction then searches the actual containing context that was specified. If the object is found there, the instruction resolves to that object. If the object is still not found an *object not found* (hex 2201) exception is signaled.

  When a containing context is specified, the current thread must have authority to each context object that is searched (e.g. the specified context object and the alias context, if any); otherwise an *unauthorized for operation* (hex 0A01) exception is signaled. If an object is found in an alias context, the specified containing context in the system ASP or basic ASPs will not be searched and authority will only be verified for the alias context (not the specified containing context object itself.)
- If the object to be resolved can reside in a context object and a containing context is *not* specified, the name resolution list is used to search for the object. The search starts with the first context object in the name resolution list. If the context object has an alias context, first the alias context is searched. Then the instruction searches the actual containing context that was specified.

  If the object still has not been found, the next context object in the the name resolution list is searched in a similar manner. This continues until all context objects in the name resolution list have been searched (or until the object is found). If the object is not found, an *object not found* (hex 2201) exception is signaled.

  For every context object in the name resolution list, the current thread must have authority to each context object that is searched (e.g. the specified context object and the alias for the context object, if any); otherwise an *unauthorized for operation* (hex 0A01) exception is signaled. For every context object in the name resolution list, if an object is found in an alias context, the specified containing context in the system ASP or basic ASPs will not be searched and authority will only verified for the alias context (not the specified containing context object itself.)

As described above, objects are located in a machine context, in a specified containing context object, or using a name resolution list of context objects. Issues regarding name spaces are now described for these cases. If an object cannot reside in an independent ASP machine context or a context object, then name spaces do not apply.

-
- Objects in a machine context (i.e. context objects)

When searching a name space for a context object, if the name space contains an ASP group and it is varied off, then that ASP group is bypassed and only the system ASP machine context is searched (for objects in the system ASP and basic ASPs).

- Objects in a context

  When searching a name space, if a containing context object is specified and that context object resides on an independent ASP that is varied off, an *object not available* (hex 220B) exception is signaled. If the containing context has an alias context and the alias context resides on an independent ASP that is varied off, then the alias context is bypassed and no exception is signaled.

- Objects in context searched using name resolution list

  If a name resolution list is used to find an object, if the context object referenced by the name resolution list entry is destroyed or it resides on an independent ASP that is varied off, that context name is bypassed and the search continues with the next context object in the name resolution list. If the context object referenced by the name resolution list entry has an alias context and the alias context resides on an independent ASP that is varied off, then the alias context is bypassed and no exception is signaled.

The following describes the instruction's function when operand 2 is null (or if the operand 2 template is extended and the field *ignore object specification and authorization fields* is set to 1). Note that operand 3 is always ignored in this case.

-
- If operand 1 does not contain a system pointer, an exception is signaled.
- If the system pointer specified by operand 1 is not resolved but has an initial value declaration, the instruction resolves the system pointer to the object that the initial value describes. The initial value defines the following:
  -
  - Object to be located (by *type code*, *subtype code*, and *object name*)
  - Name of *context* to be searched to locate the object (optional)
  - Minimum *required authorization* required for the object

  If a *context* name is specified, then that context will be searched to locate the object. If no context is specified, the context object(s) located by the name resolution list associated with the thread issuing this instruction is used to locate the object. In both cases, the current thread's name space will be used (unless the *search method* field in the extended template is used to specify a different method).

  If the minimum *required authorization* in the initial value is not set (binary 0), the instruction resolves the operand 1 system pointer to the first object encountered with the designated type code, subtype code, and object name without regard to the authorization available to the thread for the object. If one or more authorization (or ownership) states are required (signified by binary 1's), the context(s) is searched until an object is encountered with the designated type, subtype, and name for which the thread currently has all required authorization states.
- If the system pointer specified by operand 1 is currently resolved to address an existing object, the instruction does not modify the addressability contained in the pointer and causes only the authority attribute in the pointer to be modified based on operand 4.

If operand 2 is not null, then the object identified by operand 2 is resolved. (But if the operand 2 template is extended and the field *ignore object specification and authorization fields* is set to binary 1, then the object identified by operand 2 is ignored. See the preceding section, starting with the phrase "The following describes the instruction's function when operand 2 is null...".) When the object identified by operand 2 is resolved, the instruction searches the context(s) specified by operand 3 and stores the resolved system pointer in operand 1.

The format of operand 2 is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Object specification	Char(32)	
0	0		Type code	Char(1)
1	1		Subtype code	Char(1)
2	2		Object name	Char(30)
32	20	Required authorization (1 = required)	Char(2)	
32	20		Object control	Bit 0
32	20		Object management	Bit 1
32	20		Authorized pointer	Bit 2
32	20		Space authority	Bit 3
32	20		Retrieve	Bit 4
32	20		Insert	Bit 5
32	20		Delete	Bit 6
32	20		Update	Bit 7
32	20		Ownership	Bit 8
32	20		Excluded	Bit 9
32	20		Authority list management	Bit 10
32	20		Execute	Bit 11
32	20		Alter	Bit 12
32	20		Reference	Bit 13
32	20		Reserved (binary 0)	Bit 14
32	20		Extended template	Bit 15
			0 = The template is not extended	
			1 = The template is extended	
34	22	—- End —-		

The allowed *type code*s are as follows:

Hex 01 = Access group

Hex 02 = Program

Hex 03 = Module

Hex 04 = Context

Hex 06 = Byte string space

Hex 07 = Journal space

Hex 08 = User profile

Hex 09 = Journal port

Hex 0A = Queue

Hex 0B = Data space

Hex 0C = Data space index

Hex 0D = Cursor

Hex 0E = Index

Hex 0F = Commit block

Hex 10 = Logical unit description

Hex 11 = Network description

Hex 12 = Controller description

Hex 13 = Dump space

Hex 14 = Class of service description

Hex 15 = Mode description

Hex 16 = Network interface description

Hex 17 = Connection list

Hex 18 = Queue space

Hex 19 = Space

Hex 1A = Process control space

Hex 1B = Authority list

Hex 1C = Dictionary

Hex 1D = Auxiliary server

Hex 1E = Byte stream file

Hex 21 = Composite object group

Hex 23 = Transaction control structure

All other codes are reserved. If other codes are specified, they cause a *scalar value invalid* (hex 3203) exception to be signaled. This instruction will not resolve to hidden contexts. An attempt to do so will always result in an *object not found* (hex 2201) exception. A hidden context is denoted by the *hidden* attribute of a context. See the MATCTX instruction for additional details.

When resolving to an object that can reside in a context object, operand 3 identifies the context in which to locate the object identified by operand 2. If operand 3 is null, then the contexts identified in the name resolution list associated with the thread issuing this instruction are searched in the order in which they appear in the list. If operand 3 is not null, the system pointer specified must address a context, and only this context (and possible aliases for the context) are used to locate the object.

If the **required authorization** field in operand 2 is not set (all values set to 0), the instruction resolves the operand 1 system pointer to the first object encountered with the designated **type code**, **subtype code**, and **object name** without regard to the authorization currently available to the thread. If one or more authorization (or ownership) states are required (signified by binary 1's), the context is searched until an object is encountered with the designated *type code, subtype code, object name*, and the user profiles governing the thread's execution that have all the required authorization states.

If the bit **extended template** is set to binary 1, then the extended template is defined starting at offset 34 of the operand 2 template. The extended template is defined as follows:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
34	22	Template version	Char(1)
35	23	Search method	Char(1)

Offset Dec	Hex	Field Name	Data Type and Length	
		Hex 00 = Search the thread's name space		
		Hex 01 = Search only for objects residing on the specified independent ASP		
		Hex 02 = Search only for objects residing on the ASP group containing the specified independent ASP		
		Hex 03 = Search only the ASP group of the name space		
36	24	Options	Char(1)	
36	24		Ignore object specification and authorization fields	Bit 0
			1 = Ignore these values	
			0 = Use these values	
36	24		Reserved (binary 0)	Bits 1-7
37	25	Reserved (binary 0)	Char(7)	
44	2C	Independent ASP number to search	Char(2)	
46	2E	ASP number of context containing resolved object	Char(2)	
48	30	Context containing the resolved object	System pointer	
64	40	Reserved (binary 0)	Char(64)	
128	80	—- End —-		

The field **template version** identifies the version of the extended template. It must be set to hex 00.

The field **search method** determines how the name space is searched:

•

- A value of hex 00 means the thread's name space is searched, as described above (where the term "name space" is defined).
- A value of hex 01 means that the the search is restricted to objects that reside on a specified independent ASP (or the system ASP and basic ASPs). The field *independent ASP number to search* determines which ASP. This option is applicable when searching a specified context object or when searching the name resolution list. When the object resides in an ASP group, this instruction will search context objects in all independent ASPs of the ASP group.

  The field *independent ASP number to search* must specify an existing independent ASP. A value of 0 specifies the system ASP and basic ASPs. Values 33 through 255 specify an independent ASP. An ASP number between 1 and 32 or a number greater than 255 results in a *template value invalid* (hex 3801) exception being signaled. If the specified independent ASP is not varied on, then a *object not available* (hex 220B) exception is signaled.
- A value of hex 02 means that the search is restricted to objects that reside on a single ASP group or the system ASP and basic ASPs. The field *independent ASP number to search* determines which one. A value of 0 specifies the system ASP and basic ASPs. Values 33 through 255 specify the ASP group or UDFS (User Defined File System) ASP containing that independent ASP. (Any independent ASP in an ASP group can be used to identify that ASP group.) This option is applicable when searching a specified context object or when searching the name resolution list. The rules for the field **independent ASP number to search** are the same as those given for *search method* hex 01.

  A User Defined File System (UDFS) ASP is an IASP that is never part of an ASP group.

- A value of hex 03 means the search is restricted to the ASP group of the name space (not the system ASP or basic ASPs).

As stated above, if a resolve is done for an object type that can only be addressed through the system ASP machine context then only the system machine context is searched. When resolving to objects that can only reside in the system ASP machine context, the field *search method* must be set to hex 00 (if the template is extended). Otherwise a *template value invalid* (hex 3801) exception is signaled.

For option hex 01 or hex 02, if a context address is provided in operand 3, then the context object must reside in the same ASP group as the specified *independent ASP to search*; otherwise an *auxiliary storage pool number invalid* (hex 1C09) exception is signaled.

For option hex 01 or hex 02, execute authority is required for the device description for every independent ASP in the ASP group (the ASP group that contains the *independent ASP number to search*). Authority to a device description is checked only when the corresponding independent ASP is searched. Once an object is found, the remaining independent ASPs do not have to be searched.

For option hex 03, if a context address is provided in operand 3, then it must reference a context object in an ASP group of the current thread's name space or in the system ASP or basic ASP; otherwise an *auxiliary storage pool number invalid* (hex 1C09) exception is signaled. If the context address refers to a context object in the system ASP or basic ASP, then only the alias contexts are searched.

When the field **ignore object specification and authorization fields** is set to binary 0, the *object specification* and *required authorization* will be used as defined above. If this field is set to binary 1, then the *object specification* and *required authorization* fields will be ignored, as if operand 2 were a null pointer.

When an object is successfully resolved by this instruction, the field **context containing the resolved object** will contain a pointer to the context object in which the object was found. A null pointer value means the object was found in a machine context. The field **ASP number of context containing resolved object** indicates the ASP number on which the *context containing the resolved object* resides. This will be a basic ASP number or an independent ASP number. When the field *ASP number of context containing resolved object* is set to hex 00, it identifies the system ASP.

Once addressability has been set in the operand 1 pointer, operand 4 is used to determine which, if any, of the object authority states is to be set into the pointer. Only the object authority states correlating with bits 0 through 7 and 11, that is, *object control* through *update* and *execute*, can be set into the pointer. This restriction applies whether the authority mask controlling which authorities to set in the pointer comes from operand 4, operand 2, or the initial value for the system pointer.

If operand 4 is null, the object authority states required to locate the object are set in the operand 1 pointer. This required object authority is as specified in operand 2 or in the initial value for operand 1 if operand 2 is null (or if the operand 2 template is extended and the field *ignore object specification and authorization fields* is set to 1). If the thread does not currently have authorized pointer authority for the object, no authority is stored in the system pointer, and no exception is signaled.

If operand 2 is null (or if the operand 2 template is extended and the field *ignore object specification and authorization fields* is set to 1) and operand 4 is null and operand 1 is a resolved system pointer, the authority states in the pointer are not modified.

If operand 4 is not null, it specifies the object authority states to be set in the operand 1 system pointer. The format of operand 4 is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Requested authorization (1 = set authority)	Char(2)	
0	0		Object control	Bit 0

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0		Object management	Bit 1
0	0		Authorized pointer	Bit 2
0	0		Space authority	Bit 3
0	0		Retrieve	Bit 4
0	0		Insert	Bit 5
0	0		Delete	Bit 6
0	0		Update	Bit 7
0	0		Reserved (binary 0)	Bits 8-10
0	0		Execute	Bit 11
0	0		Alter (will be ignored)	Bit 12
0	0		Reference (will be ignored)	Bit 13
0	0		Reserved (binary 0)	Bits 14-15
2	2	—- End —-		

The authority states set in the operand 1 system pointer are based on the following:

•

- The authority already stored in the pointer can be increased only when the thread has authorized pointer authority to the referenced object. If this authority is not available and the pointer was resolved by this instruction, the authority in the operand 1 system pointer is set to the not set state, and no exception is signaled. If operand 2 is null (or if the operand 2 template is extended and the field *ignore object specification and authorization fields* is set to 1), if operand 1 is a resolved system pointer containing authority, and if authorized pointer authority is not available to the thread, additional authorities cannot be stored in the pointer.
- If the thread does not currently have all the authority states requested in operand 4, only the requested and available states are set in the pointer, and no exception is signaled.
- A thread executing in user state may not set any additional authority in a system pointer. Operand 4 will be ignored if the thread executing this instruction is running in user state.
- Note that the authority stored in the operand 1 system pointer is a source of authority applies to this instruction when operand 2 is null (or if the operand 2 template is extended and the field *ignore object specification and authorization fields* is set to 1) and operand 1 is a resolved system pointer with authority stored in it.

Warning: Temporary Level 3 Header

Authorization Required

•

- Execute
  –
  – Contexts referenced for address resolution. This includes the operand 3 system pointer (when it is not a null system pointer) and system pointers obtained from the name resolution list. Authority is

also checked for aliases of context objects. The authority supplied in the actual pointers is used when verifying authority for each context (referenced by the operand 3 or the name resolution list) and for any alias context.

– Device description for every independent ASP in a ASP group when a resolve is done for an object on a specified independent ASP or ASP group. See the description of the *search method* field for details.

Lock Enforcement

•

• Materialization

  –

  – Contexts referenced for address resolution (including operand 3)

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C09 Auxiliary Storage Pool Number Invalid

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2404 Pointer Not Resolved

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Footnotes:

[1] Programs executing in user-state may not assign authority in the resulting system pointer. The value in operand 4 is ignored and no exception is raised.

Retrieve Computational Attributes (RETCA)

Bound program access

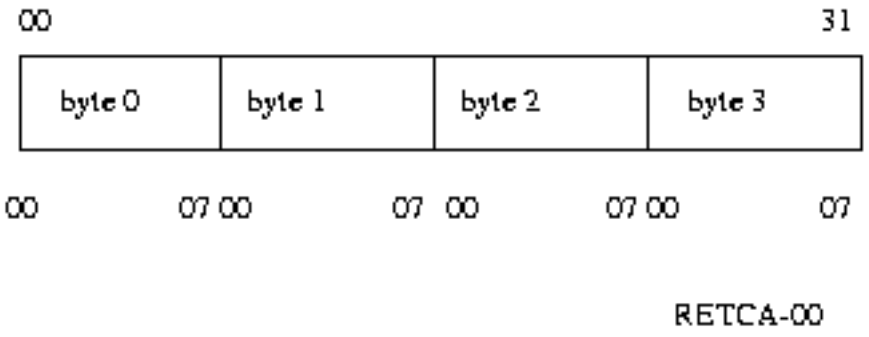
```
Built-in number for RETCA is 145.
RETCA (
       selector   : unsigned binary(4) literal value; its rightmost byte
                    specifies the computational attributes to retrieve
) :   unsigned binary(4) value which contains the computational attributes
      specified by selector
```

Description: The right-most byte of *selector* specifies the computational attributes to retrieve. The format of this byte is as follows:

Bit	Definition
0-3	Reserved (must be 0)
4	Exception mask
5	Reserved (must be 0)
6	Exception occurrence
7	Rounding mode

All other bytes of *selector* are reserved (must be 0).

The value returned by RETCA has the following structure:



•

• Byte 0: Exception mask

Bit	Meaning
0-1	Reserved (binary 0)
2	Floating-point overflow
3	Floating-point underflow
4	Floating-point zero divide
5	Floating-point inexact result
6	Floating-point invalid operand
7	Reserved (binary 0)

• Byte 1: Reserved (binary 0)

• Byte 2: Exception occurrence

Bit	Meaning
0-1	Reserved (binary 0)
2	Floating-point overflow
3	Floating-point underflow
4	Floating-point zero divide

Bit	Meaning
5	Floating-point inexact result
6	Floating-point invalid operand
7	Reserved (binary 0)

- Byte 3: Computational mode

Bit	Meaning
0	Reserved (binary 0)
1-2	Rounding mode • 00 - Round towards positive infinity • 01 - Round towards negative infinity • 10 - Round towards 0 • 11 - Round to nearest
3-7	Reserved (binary 0)

Note: Any floating-point operations currently on the value stack will be computed prior to retrieving the computational attributes. Therefore, the effect of such floating-point operations on the *exception occurrence* byte, for example, will be reflected in the value returned by the function.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

•

- None

Retrieve Exception Data (RETEXCPD)

Op Code (Hex)	Operand 1	Operand 2
03E2	Receiver	Retrieve options

Operand 1: Space pointer.

Operand 2: Character(1) scalar.

Description: The data related to a particular occurrence of an exception is returned and placed in the specified space.

Operand 1 is a space pointer that identifies the *receiver* template. The template identified by operand 1 must be 16-byte aligned in the space.

The value of operand 2 specifies the type of exception handler for which the exception data is to be retrieved. The exception handler may be a branch point exception handler, an internal entry point exception handler, or an external entry point exception handler.

An *exception state of thread invalid* (hex 1602) exception is signaled to the invocation issuing the Retrieve Exception Data instruction if the retrieve option is not consistent with the thread's exception handling state. For example, the exception is signaled if the retrieve option specifies retrieve for internal entry point exception handler and the thread exception state indicates that an internal exception handler has not been invoked.

After an invocation has been destroyed, exception data associated with a signaled exception description within that invocation is lost.

The format of operand 1 for the materialization is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size	Char(8)	
0	0		Number of bytes provided for retrieval	Bin(4)
4	4		Number of bytes available for retrieval	Bin(4)
8	8	Exception identification	Char(2)	
10	A	Compare value length (maximum of 32 bytes)	Bin(2)	
12	C	Compare value	Char(32)	
44	2C	Message reference key	Char(4)	
48	30	Exception specific data	Char(*)	
*	*	Source invocation	Invocation pointer or Null	
*	*	Target invocation	Invocation pointer	
*	*	Source instruction address	UBin(2)	
*	*	Target instruction address	UBin(2)	
*	*	Machine-dependent data	Char(10)	
*	*	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for retrieval of the exception data. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

The **message reference key** field returns the architected value that uniquely identifies the message in the process queue space.

The **source invocation** and **source instruction address** identify the invocation that caused the exception to be signaled. For machine exceptions, this invocation pointer identifies the invocation executing when the exception occurred. For user-signaled exceptions, this invocation pointer locates the invocation that executed the Signal Exception (SIGEXCP) instruction. The pointer will be null if the source invocation no longer exists at the time that this instruction is executed. The **source instruction address** field locates the instruction that caused the exception to be signaled. This field in a bound program invocation will be set to 0.

The **target invocation** and **target instruction address** identify the invocation that is the target of the exception. This invocation is the last invocation that was given the chance to handle the exception. For machine exceptions, the first target invocation is the invocation incurring the exception. For user-signaled exceptions, the Signal Exception (SIGEXCP) instruction may initially locate the current or any previous

invocation. If the target invocation handles the exception by resignaling the exception, the immediately previous invocation is considered to be the target invocation. This may occur repetitively until no more prior invocations exist in the thread and the signaled program invocation entry is assigned a value of binary 0. If an invocation handles the exception in any manner other than resignaling or does not handle the exception, that invocation is considered to be the target.

The *target instruction address* field specifies the number of the instruction that is currently being executed in the target invocation.

The machine extends the area beyond the *exception specific data* area with binary 0's so that the pointers to program invocations are aligned on a 16 byte boundary.

The operand 2 values are defined as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Retrieve options	Char(1)
		Hex 00 = Retrieve for a branch point exception handler	
		Hex 01 = Retrieve for an internal entry point exception handler	
		Hex 02 = Retrieve for an external entry point exception handler	
1	1	—- End —-	

If the *exception data retention* option is set to 1 (do not save), the *number of bytes available for retrieval* is set to 0.

Exception data is always available to the process default exception handler.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

16 Exception Management

1602 Exception State of Thread Invalid

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Retrieve Invocation Flags (RETINVF)

Bound program access
Built-in number for RETINVF is 147. RETINVF ( ) : unsigned binary(4) value which specifies the current invocation flags

Description: The current invocation flags are returned. The ″read-only″ flags are returned in the high-order two bytes of the result; the ″writeable″ flags are returned in the low-order two bytes of the result.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

Retrieve Teraspace Address From Space Pointer (RETTSADR)

Bound program access
Built-in number for RETTSADR is 623. RETTSADR (source_pointer : space pointer(16)) : local form address

Description: Retrieve the teraspace address from *source pointer*. If *source pointer* contains a *null pointer value* then a *null pointer value* is returned.

If *source pointer* contains anything except a space address that points to teraspace, then a *space address is not a teraspace storage address* (hex 0609) exception is signalled. If any exception is signalled during this operation the result is undefined.

This instruction may be specified in a program template only when the (bound) program is to be created as teraspace capable.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0602 Boundary Alignment

0609 Space Address Is Not A Teraspace Storage Address

24 Pointer Specification

2402 Pointer Type Invalid

Retrieve Thread Count (RETTHCNT)

Op Code (Hex)	Operand 1
0321	Count

Operand 1: Unsigned binary(4) variable scalar.

Bound program access
Built-in number for RETTHCNT is 514. RETTHCNT ( ) : unsigned binary(4) /* count */

Description: A *count* of the number of threads that have been implicitly or explicitly initiated within the process and which have not yet terminated is returned.

Since process initiation implicitly initiates a thread, the minimum value of *count* is 1.

It is possible that this instruction may be used in a cancel or return handler for a thread that is terminating. If it is, the count that will be returned will still include the terminating thread.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

36 Space Management

- 3601 Space Extension/Truncation

44 Protection Violation

- 4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Retrieve Thread Identifier (RETTHID)

Op Code (Hex)	Operand 1
0395	Thread identifier

Operand 1: Character(8) variable scalar.

Bound program access
Built-in number for RETTHID is 516. RETTHID ( ) : aggregate(8) /* thread_identifier */

Description: A *thread identifier* is returned which is unique within the process. While no two threads initiated within the same process will have the same identifier, it is possible that threads in different processes may have the same value for the identifier.

The *thread identifier* is used on some instructions to identify a thread. For example, it may be useful in associating queue space messages with a particular thread.

When a thread ends, its *thread identifier* is never reused within the process.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Return External (RTX)

Op Code (Hex)	**Operand 1**
02A1	Return point

Operand 1: Signed binary(2) scalar or null.

Description: The instruction terminates execution of the invocation in which the instruction is specified. The automatic storage frame is deallocated.

A Return External instruction can be specified within an invocation's subinvocation, and no exception is signaled.

If a higher invocation exists in the invocation hierarchy, the instruction causes execution to resume in the preceding invocation in the thread hierarchy at an instruction location indirectly specified by operand 1. If operand 1 is binary 0 or null, the next instruction following the Call External instruction from which control was relinquished in the preceding invocation in the hierarchy is given execution control. If the

value of operand 1 is not 0, the value represents an index into the instruction definition list (IDL) specified as the return list operand in the Call External instruction, and the value causes control to be passed to the instruction referenced by the corresponding IDL entry. The first IDL entry is referenced by a value of one. If operand 1 is not 0 and no return list was specified in the Call External instruction, or if the value of operand 1 exceeds the number of entries in the IDL, or if the value is negative, a *return point invalid* (hex 2C02) exception is signaled.

In the initial thread of a process, if a higher invocation does not exist, the Return External instruction causes termination of the current process state. If operand 1 is not 0 and is not null, the *return point invalid* (hex 2C02) exception is signaled.

If the returning invocation has received control to process an event, then control is returned to the point where the event handler was invoked. In this case, if operand 1 is not 0 and is not null, then a *return point invalid* (hex 2C02) exception is signaled.

If the returning invocation has received control from the machine to process an exception, the *return instruction invalid* (hex 2C01) exception is signaled.

If the returning invocation has an activation, the invocation count in the activation is decremented by 1.

If the returning invocation currently has an invocation exit set, the invocation exit is not given control and is implicitly cleared.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C01 Return Instruction Invalid

2C02 Return Point Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Return From Exception (RTNEXCP)

Op Code (Hex)	Operand 1
03E1	Return target

Operand 1: Space pointer.

Description: An internal exception handler subinvocation or an external exception handler invocation is terminated, and control is passed to the specified instruction in the specified invocation. All intervening

invocations are marked as cancelled, down to, but not including, the invocation that is being returned to. When each of these invocations are returned to, their return handlers and invocation exit (I-exit) routines/cancel handlers will be found and run.

Note: This instruction is not allowed from a bound program invocation.

The template identified by operand 1 must be 16-byte aligned in the space. It specifies the target invocation and target instruction in the invocation where control is to be passed. The format of operand 1 is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Invocation address/offset	Space pointer or Invocation pointer		
16	10	Reserved (binary 0)	Char(1)		
17	11	Action	Char(2)		
17	11		Reserved (binary 0)		Bits 0-4
17	11		Use offset option		Bit 5
			0 =	Use invocation address as a pointer value	
			1 =	Use invocation address as an offset value	
17	11		Unstack option		Bit 6
			0 =	The action performed is determined by the setting of the following action code (bit 7).	
			1 =	If the exception handler is an internal exception handler, resume execution with the instruction that follows the RTNEXCP instruction and terminate the internal exception handler subinvocation.	
17	11		Action code		Bit 7
			0 =	Re-execute the instruction that caused the exception.	
			1 =	Resume execution with the instruction that follows the instruction that caused the exception or resume execution with the instruction that follows the instruction that invoked the invocation.	
18	12		Reserved (binary 0)		Char(1)
19	13	—- End —-			

The **invocation address/offset** field is a space/invocation pointer that identifies the invocation to which control will be passed.

The target *invocation address* field can also be an offset value from the current requesting invocation to the invocation to be searched. This is done by setting the **use offset option** field that follows the *invocation address* field to 1. If the *invocation offset* value locates the invocation stack base entry, the *invocation offset outside range of current stack* (hex 2C1A) exception is signaled. If the *invocation offset* value is a positive

number (which represents newer invocations on the stack) a *template value invalid* (hex 3801) exception is signaled. The current instruction in an invocation is the one that caused another invocation to be created.

The **unstack option** is only valid when issued in an internal exception handler subinvocation and is ignored for an external exception handler invocation. This option will cause the internal exception handler subinvocation to be terminated and control will resume at the instruction immediately following the RTNEXCP instruction. In effect, this option will cause the current subinvocation to be unstacked.

If the **action code** is 0, then the current instruction of the addressed invocation is reexecuted, if it is allowed. If the *action code* is 1, execution resumes with the instruction following the current instruction of the addressed invocation, if it is allowed. If it is not, a *retry/resume invalid* (hex 1604) exception will be signaled.

For an **action code** of 0 the **invocation address/offset** field must identify the invocation which enabled the current exception handler. Otherwise a *template value invalid* (hex 3801) exception will be signalled.

The Return From Exception instruction may be issued only from the initial invocation of an external exception handling sequence or from an invocation that has an active internal exception handler.

If the instruction is issued from an invocation that is not an external exception handler and has no internal exception handler subinvocations, the *return instruction invalid* (hex 2C01) exception is signaled.

The following table shows the actions performed by the Return From Exception instruction:

Invocation Issuing Instruction	Addressing Own Invocation/Option	Addressing Higher Invocation/Option
Not handling exception	Error (see note 1)	Error (see note 1)
Handling internal exception(s)	Allowed (see note 2)	Allowed (see note 3)
Handling external exception(s)	Error (see note 1)	Allowed (see note 3)
Handling external exception(s) and internal exception(s)	Allowed (see note 2)	Allowed (see note 3)

Notes::

1. A *return instruction invalid* (hex 2C01) exception is signaled. If there are no more internal exception handler subinvocations active and this invocation is not an external exception handler, the instruction may not be issued.
2. The current internal exception handler subinvocation is terminated.
3. All invocations after the addressed invocation are terminated and execution proceeds within the addressed invocation. Any invocation exit programs set for the terminated invocations will be given control before execution proceeds within the addressed invocation. This option is only allowed when the action code specified is a 1.

Whenever an invocation is terminated, the invocation count in the corresponding activation entry (if any) is decremented by 1.

An *action code* of 1 specifies completion of an instruction rather than execution of the following instruction if the current instruction in the addressed invocation signaled a *size* (hex 0C0A) exception or a *floating-point inexact result* (hex 0C0D) exception.

Note: The previous condition does not apply if any of the above exceptions were explicitly signaled by a Signal Exception (SIGEXCP) instruction.

A Return From Exception instruction cannot be used or recognized in conjunction with a branch point internal exception handler.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

16 Exception Management

1603 Invalid Invocation Address

1604 Retry/Resume Invalid

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C01 Return Instruction Invalid

2C12 Activation Group Access Violation

2C1A Invocation Offset Outside Range of Current Stack

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Return PCO Pointer (PCOPTR2)

Bound program access

`Built-in number for PCOPTR2 is 358.`
`PCOPTR2 (`
`) :  space pointer(16) to the process' PCO (process communication object)`

Description: Return PCO Pointer (PCOPTR2) obtains addressability to a process' PCO (process communication object) and returns it in a space pointer.

This built-in function supports a high performance alternative method of obtaining the PCO pointer of the currently executing process.

Note: Another alternative for obtaining the PCO pointer is via option hex 17 on the Materialize Process Attributes (MATPRATR) instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

36 Space Management

3601 Space Extension/Truncation

Scale (SCALE)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
SCALE 1063		Receiver	Source	Scale factor	
SCALEI 1863	Indicator options	Receiver	Source	Scale factor	Indicator targets
SCALEB 1C63	Branch options	Receiver	Source	Scale factor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Binary(2) scalar.

Operand 4-7:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
SCALES 1163		Receiver/Source	Scale factor	
SCALEIS 1963	Indicator options	Receiver/Source	Scale factor	Indicator targets
SCALEBS 1D63	Branch options	Receiver/Source	Scale factor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Binary(2) scalar.

Operand 3-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The scale instruction performs numeric scaling of the source *operand* based on the scale factor and places the results in the *receiver* operand. The numeric operation is as follows:

```
Operand 1 = Operand 2 *(B**N)
```

where:

N is the binary integer value of the scale operand. It can be positive, negative, or 0. If N is 0, then the operation simply copies the *source* operand value into the *receiver* operand.

B is the arithmetic base for the type of numeric value in the *source* operand.

Base Type	**B**
Binary	2
Packed/Zoned	10
Floating-point	2

The operands must be of the numeric types indicated with any implicit conversions occurring according to the rules of arithmetic operations as outlined in the Arithmetic Operations. The scale operation is a shift of N unsigned binary, packed, or zoned digits. The shift is to the left if N is positive, to the right if N is negative. For a signed binary *source* operand, the scale operation is performed as if the *source* operand is multiplied by a signed binary value of 2**N.

For a floating-point *source* operand, the scale operation is performed as if the *source* operand is multiplied by a floating-point value of 2**N.

If the *source* and *receiver* operands have different attributes, the scaling operation is done in an intermediate field with the same attributes as the *source* operand. If a fixed-point scaling operation causes nonzero digits to be truncated on the left end of the intermediate field, a *size* (hex 0C0A) exception is signaled. For a floating-point scaling operation, the *floating-point overflow* (hex 0C06) exception and the *floating-point underflow* (hex 0C07) exception can be signaled during the calculation of the intermediate result.

The resultant value of the calculation is copied into the *receiver* operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the *receiver* operand, aligned at the assumed decimal point of the *receiver* operand, or both before being copied to it. Length adjustment and decimal point alignment are performed according to the rules of arithmetic operations outlined in Arithmetic Operations. For fixed-point operations, if nonzero digits are truncated off the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For floating-point operations involving fixed-point receiver fields, if nonzero digits would be truncated from the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For floating-point *receiver* fields, if the exponent of the resultant value is either too large or too small to be represented in the *receiver* field, the *floating-point overflow* (hex 0C06) exception or *floating-point underflow* (hex 0C07) exception is signaled.

A *scalar value invalid* (hex 3203) exception is signaled if the value of N is beyond the range of the particular type of the *source* operand as specified in the following table.

Source Operand Type	**Range of N**
Signed Binary(2)	-14 <= N <= 14
Unsigned Binary(2)	-15 <= N <= 15
Signed Binary(4)	-30 <= N <= 30
Unsigned Binary(4)	-31 <= N <= 31

Source Operand Type	Range of N
Decimal(P,Q)	-31 <= N <= 31

For a scale operation in floating-point, no limitations are placed on the values allowed for N other than the implicit limits imposed due to the floating-point overflow and underflow exceptions.

Limitations (Subject to Change): The following are limits that apply to the functions performed by this instruction.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Condition:

•

- Positive-The algebraic value of the *receiver* operand is positive.
- Negative-The algebraic value of the *receiver* operand is negative.
- Zero-The algebraic value of the *receiver* operand is zero.
- Unordered-The value assigned a floating-point *receiver* operand is NaN.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Scan (SCAN)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
SCAN 10D3		Receiver	Base	Compare operand	
SCANB 1CD3	Branch options	Receiver	Base	Compare operand	Branch targets
SCANI 18D3	Indicator options	Receiver	Base	Compare operand	Indicator targets

Operand 1: Binary variable scalar or binary array.

Operand 2: Character variable scalar.

Operand 3: Character scalar.

Operand 4-5:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The character string value of the *base* operand is scanned for occurrences of the character string value of the *compare* operand.

The *base* and *compare* operands must both be character strings. The length of the *compare* operand must not be greater than that of the base string.

The operation begins at the left end of the base string and continues character by character, from left to right, comparing the characters of the base string with those of the *compare* operand. The length of the comparisons are equal to the length of the *compare* operand value and function as if they were being compared in the Compare Bytes Left-Adjusted (CMPBLA)instruction.

If a set of bytes that match the *compare* operand is found, the binary value for the ordinal position of its leftmost base string character is placed in the *receiver* operand.

If the *receiver* operand is a scalar, only the first occurrence of the *compare* operand is noted. If it is an array, as many occurrences as there are elements in the array are noted.

The operation continues until no more occurrences of the *compare* operand can be noted in the *receiver* operand or until the number of characters (bytes) remaining to be scanned in the base string is less than the length of the *compare* operand.

When the second condition occurs, the *receiver* value is set to 0. If the *receiver* operand is an array, all its remaining elements are also set to 0.

The *base* operand and the *compare* operand can be variable length substring compound operands.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 2 and 3. The effect of specifying a null substring reference for the *compare* operand or both operands is that the *receiver* is set to zero (no match found) and the instruction's resultant

condition is null *compare* operand. Specifying a null substring reference for just the *base* operand is not allowed due to the requirement that the length of the *compare* operand must not be greater than that of the base string.

Resultant Conditions:

•

- Zero-The numeric value(s) of the receiver *operand* is zero. When the *receiver* operand is an array, the resultant condition is zero if all elements are zero. One of these two conditions will result when the *compare* operand is not a null substring reference.
- Positive-The numeric value(s) of the *receiver* operand is positive.
- Null compare operand-The compare operand is a null substring reference; therefore, the *receiver* has been set to zero which indicates that no occurrences were found.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C08 Length Conformance

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Scan Extended (SCANX)

Bound program access
Built-in number for SCANX is 415. SCANX (base_locator : address of a space pointer(16) base locator scan_controls : address of scan controls scan_options : literal(4) containing scan options) : signed binary(4) value to indicate the manner in which the instruction completed

Description: The base string to be scanned is specified by the *base locator* and controls operands. The *base locator* addresses the first character of the base string. The controls specifies the length of the base string in the *base length* field.

The scan operation begins at the left end of the base string and continues character by character, left-to-right. The scan operation can be performed on a base string which contains all simple (1-byte) or all extended (2-byte) character codes or a mixture of the two. When the base string is being interpreted in simple character mode, the operation moves through the base string one byte at a time. When the base string is being interpreted in extended character mode, the operation moves through the base string 2 bytes at a time. The character string value of the base operand is scanned for occurrences of a character value satisfying the criteria specified in the control and options operands.

The scan is completed by updating the *base locator* and controls operands with scan status when a character value being scanned for is found, the end of the base string is encountered, or an escape code is encountered when the *test for escape codes* option is specified within the *scan controls* operand. A completion code indicating the manner in which the instruction completed is also returned. The *base locator* is set with addressability to the character (simple or extended) which caused the instruction to complete execution. The controls operand is set with information which identifies the mode (simple or extended) of the base string character addressed by the *base locator* and which provides for resumption of the scan operation with minimal overhead.

The controls and options operands specify the modes to be used in interpreting characters during the scan operation. Characters can be interpreted in one of two character modes: simple (1-byte) and extended (2-byte). Additionally, the base string can be scanned in one of two scan modes, mixed (base string may contain a mixture of both character modes) and nonmixed (base string contains one mode of characters).

When the mixed scan mode is specified in the options operand, the base string is interpreted as containing a mixture of simple and extended character codes. The mode, simple or extended, with which the string is to be interpreted, is controlled initially by the base mode indicator in the controls operand and thereafter by mode control characters imbedded in the base string. The mode control characters are as follows:

•
- Hex 0E = Shift out (SO) of simple character mode to extended mode.
- Hex 0F = Shift in (SI) to simple character mode from extended mode. This is only recognized if it occurs in the first byte position of an extended character code.

When the nonmixed scan mode is specified in the options operand, the base string is interpreted using only the character mode specified by the base mode indicator in the controls operand. Character mode shifting can not occur because no mode control characters are recognized when scanning in nonmixed mode.

The *base locator* operand is a space pointer which is both input to and output from the instruction. On input, it locates the first character of the base string to be processed. On output, it locates the character of the base string which caused the instruction to complete.

The controls operand is the address of an aggregate which specifies additional information to be used to control the scan operation. The aggregate *scan controls* must be at least 8 bytes long and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Scan controls	Char(24)	
0	0		Control indicators	Char(1)
0	0		Base mode	Bit 0
			0 = Simple	
			1 = Extended	
0	0		Comparison character mode	Bit 1

Offset Dec	Offset Hex	Field Name	Data Type and Length		
			0 =	Simple	
			1 =	Extended	
0	0		Reserved		Bits 2-5
0	0		Enhanced options		Bit 6
			0 =	Enhanced options fields are not used	
			1 =	Enhanced options fields are used	
0	0		Scan state		Bit 7
			0 =	Resume scan	
			1 =	Start scan	
1	1		Ignored		Char(1)
2	2		Comparison character		Char(2)
4	4		Reserved (binary 0)		Char(1)
5	5		Base end		Char(3)
5	5		Instruction work area		Char(1)
6	6		Base length		Char(2)
8	8		Enhanced length		UBin(8)
16	10		Enhanced resume info		UBin(8)
24	18	—- End —-			

Only the first 8 or 24 bytes of *scan controls* are used, depending upon the value of *enhanced options*. Any excess bytes are ignored.

The **base mode** is both input to and output from the instruction. In either case, it specifies the mode of the character in the base string currently addressed by the *base locator*.

The **comparison character mode** is not changed by the instruction. It specifies the mode of the comparison character contained in the controls operand.

The **scan state** is both input to and output from the instruction. As input, it indicates whether the scan operation for the base string is being started or resumed. If it is being *started*, the instruction assumes that the *base length* value in the *base end* field of the controls operand specifies the length of the base string, and the instruction work area value is ignored. If it is being *resumed*, the instruction assumes the *base end* field has been set by a prior start scan execution of the instruction with an internal machine value identifying the end of the base string.

For a *start scan* execution of the instruction, the *scan state* field is reset to indicate resume scan to provide for subsequent resumption of the scan operation. Additionally, for a *start scan* execution of the instruction, the *base end* field is set with an internally optimized value which identifies the end of the base string being scanned. This value then overlays the values which were in the instruction work area and base length fields on input to the instruction. Predictable operation of the instruction on a *resume scan* execution depends upon this base end field being left intact with the value set by the start scan execution.

For a *resume scan* execution of the instruction, the *scan state* and *base end* fields are unchanged.

The **comparison character** is input to the instruction. It specifies a character code to be used in the comparisons performed during the scanning of the base string. The *comparison character mode* in the

control indicators specifies the mode (simple or extended) of the comparison character. If it is a *simple* character, the first byte of the *comparison character* field is ignored and the *comparison character* is assumed to be specified in the second byte. If it is an *extended* character, the *comparison character* is specified as a 2-byte value in the *comparison character* field.

When **enhanced options** has a value of 0, the *base end* value is used. Otherwise the *enhanced length* and *enhanced resume info* fields are used and *base length is ignored*. The value of *enhanced options* must not be changed between *start scan* and *resume scan* executions on the same string.

When *base locator* points to a space pointer which contains a teraspace address, an *unsupported space use* (hex 0607) exception is signaled if *enhanced options* has a value of 0 and *resume scan* is specified.

The **base end** field is both input to and output from the instruction. It contains data which identifies the end of the base string. Initially, for a start scan execution of the instruction, it contains the length of the base string in the **base length** field. Additionally, the *base end* field is used to retain information over multiple instruction executions which provides for minimizing the overhead required to resume the scan operation for a particular base string. This information is set on the initial *start scan* execution of the instruction and is used during subsequent *resume scan* executions of the instruction to determine the end of the base string to be scanned. If the end of the base string being scanned must be altered during iterative usage of this instruction, a *start scan* execution of the instruction must be performed to provide for correctly resetting the internally optimized value to be stored in the *base end* from the values specified in the *base locator* operand and *base length* field.

The **enhanced length** field is input to the instruction. It contains the length in bytes of the string to be scanned when *enhanced options* has a value of 1. Current machine implementations support a maximum length of 16777215; larger values cause a *scalar value invalid* (hex 3203) exception to be signaled.

The **enhanced resume info** field is both input to and output from the instruction but is only used when *enhanced options* has a value of 1. This field is set with internal information during a *start scan* execution of this instruction and used as input for subsequent *resume scan* executions of this instruction.

If the end of the base string being scanned must be altered during iterative usage of this instruction, a *start scan* execution of the instruction must be performed to provide for correctly resetting the internally optimized value to be stored in *enhanced resume info* from the values specified in the *base locator* operand and *enhanced length* field.

For the special case of a *start scan* execution where a length value of zero (no characters to scan) is specified in either the *base length* field when *enhanced options* has a value of 0 or in the *enhanced length* field when *enhanced options* has a value of 1, the instruction results in a *not found* resultant condition. In this case, the base string is not verified and the *scan state* indicator, the *base end* field, and the *base locator* are not changed.

The options operand must be a literal which specifies the options to be used to control the scan operation. *Scan options* must be at least 4 bytes in length and has the following format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Scan options	Char(4)	
0	0		Options indicators	Char(1)
1	1		Reserved (binary 0)	Char(3)
4	4	—- End —-		

The **option indicators** field has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Option indicators	Char(1)	
0	0		Reserved (binary 0)	Bit 0
0	0		Scan mode	Bit 1
			0 = Mixed	
			1 = Nonmixed	
0	0		Reserved	Bits 2-3
0	0		Comparison relation	Bits 4-6
0	0		Equal, (=) relation	Bit 4
0	0		Less than, (<) relation	Bit 5
0	0		Greater than, (>) relation	Bit 6
			0 = No match on relation	
			1 = Match on relation	
0	0		Test for escape codes	Bit 7
			0 = Do not test for escape codes during the scan	
			1 = Test for escape codes during the scan	
1	1	—- End —-		

The **scan mode** specifies whether the base string contains a mixture of character modes, or contains all one mode of characters; that is, whether or not mode control characters should be recognized in the base string. *Mixed* specifies that there is a mixture of character modes and, therefore, mode control characters should be recognized. *Nonmixed* specifies that there is not a mixture of character modes and, therefore, mode control characters should not be recognized. Note that the *base mode* indicator in the controls operand specifies the character mode of the base string character addressed by the *base locator*.

The **comparison relation** specifies the relation or relations of the comparison character to characters of the base string which will satisfy the scan operation and cause completion of the instruction with one of the *high, low, or equal* resultant conditions. Multiple relations may be specified in conjunction with one another. Specifying all relations insures a match against any character in the base string which is of the same mode as the comparison character. Specifying no relation insures a *not found* resultant condition, unless the instruction is testing for escape codes and an escape code value is found, regardless of the values of the characters in the base string which match the mode of the comparison character.

An example of comparison scanning is a scan of simple mode characters for a value less than hex 40. This could be done by specifying a *comparison character* of hex 40 and a *comparison relation* of *greater than*. This could also be done by specifying a *comparison character* of hex 3F and *comparison relations* of *equal* and *greater than*.

The **test for escape codes** field determines whether the base string is tested for values less than hex 40 while the scan is being performed. This testing, if requested, is always performed in conjunction with whatever comparison processing has been requested. That is, escape code testing is performed even if no comparison relation is specified. The following material discusses this function in more detail.

Operation: During the scan operation, the characters of the base string which are not of the same mode as the comparison character are skipped over until the mode of the characters being processed is the same as the mode of the comparison character. The operation then proceeds by comparing the comparison character with each of the characters of the base string.

If a base string character satisfying the criteria specified in the controls and options operands is found, the *base locator* is set to address the first byte of it, the *base mode* indicator is set to indicate the mode of the base string as of that character, and the instruction is completed with the appropriate completion code, based on the *comparison relation* (*high, low*, or *equal*) of the *comparison character* to the base string character.

If a matching base string character is not found prior to encountering a mode change, the characters of the base string are again skipped over until the mode of the characters being processed is the same as the mode of the comparison character before comparisons are resumed.

If a matching base string character is not found prior to encountering the end of the base string, the base location is set to address the first byte of the character encountered at the end of the base string, the *base mode* indicator is set to indicate the mode of the base string as of that character, and the instruction is completed with the *not found* completion code. A mode control string results in the changing of the base string mode, but the *base locator* is left addressing the mode control character.

If *test for escape codes* has a value of 1, the test is performed on the characters of the base string prior to their being skipped or compared with the comparison character. Each byte of the base string is checked for a value less than hex 40. Additionally, for a *mixed* scan mode, when such a value is encountered, it is then determined if it is a valid mode control character.

•
- Hex 0E (S0) when the base string is being interpreted in simple character mode.
- Hex 0F (SI) in the left byte of the character code when the base string is being interpreted in extended character mode.

If a byte value of less than hex 40 is not a valid mode control character, it is considered to be an escape code. The *base locator* is set to address the first byte of the base string character (simple or extended) which contains the escape code, the *base mode* indicator is set to indicate the mode of the base string as of that character, and the *completion code* is set to indicate that an escape code was found.

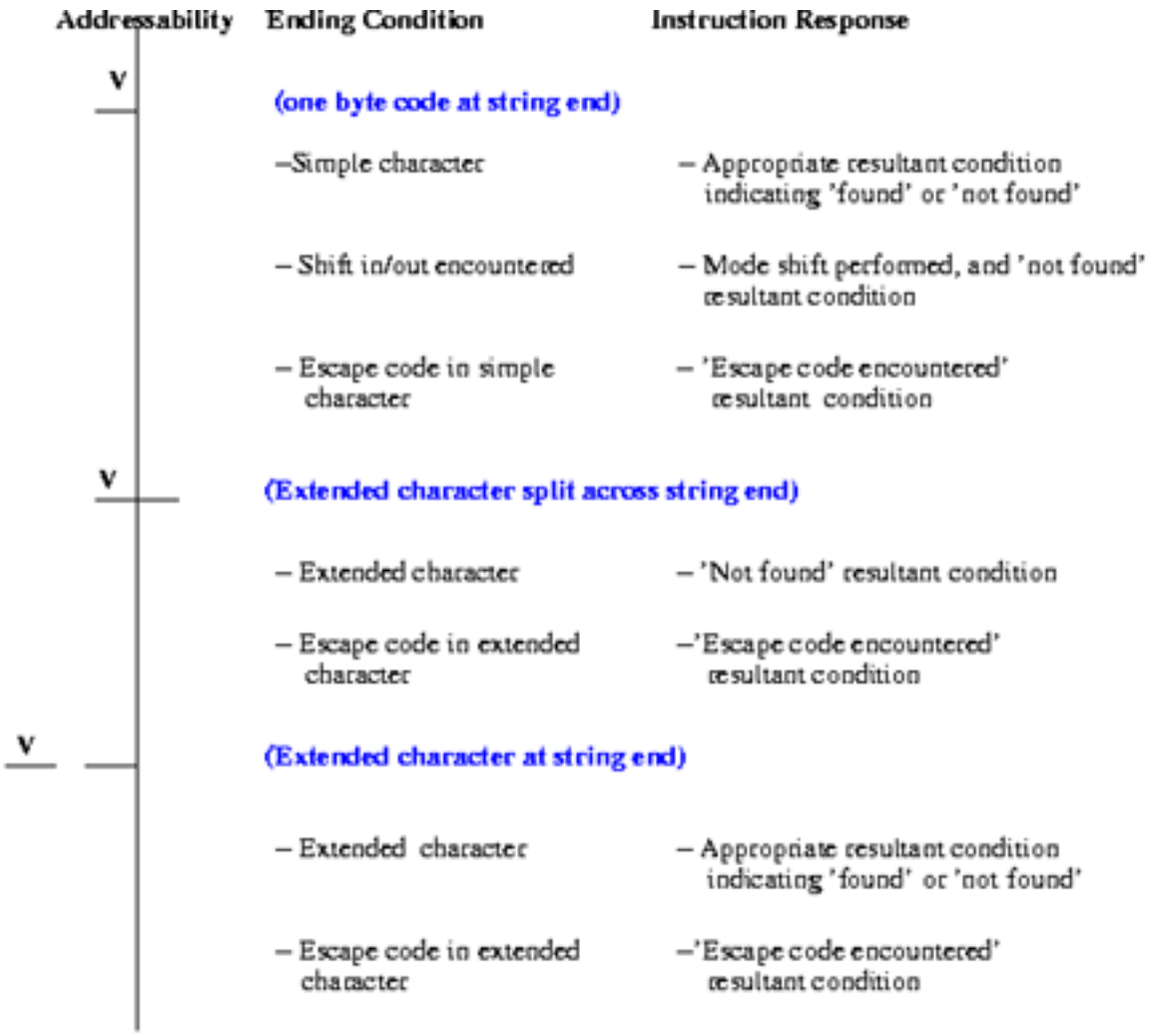
If possible, specify *scan controls* on an 8-byte multiple (doubleword) boundary relative to the start of the space containing it. Appreciably less overhead is incurred in accessing and storing the value of the controls if this is done.

For the case where a base string is to be just scanned for byte values less than hex 40, two techniques can be used.

•
- A direct simple mode scan for a value less than hex 40 without usage of the *test for escape codes* feature.
- A scan for any character with usage of the *test for escape codes* feature.

  The direct scan approach, the former, is the more efficient.

The following diagram defines the various conditions which can be encountered at the end of the base string and what the *base locator* addressability is in each case. The solid vertical line represents the end of the base string. The dashes represent the bytes before and after the base string end. The V is positioned over the byte addressed by the *base locator* in each case. These are the conditions which can be encountered when the *base locator* input to the instruction addresses a byte prior to the base string end. When the base length field specifies a value of zero for a start scan execution of the instruction, or the input *base locator* addresses a point beyond the end of the instruction, no processing is performed and the instruction is immediately completed with the *not found* completion code value.



An analysis of the diagram shows that normally, after appropriate processing for the particular *found, not found, or escape* condition, the scan can be restarted at the byte of data which would follow the base string end in the data stream being scanned. Any mode shift required by an ending mode control character will have been performed.

However, one ending condition may require subsequent resumption of the scan at the character encountered at the end of the base string. This is the case where the instruction completes with the *not found* completion code value and the base string ends with an extended character split across string end. That is, the *base mode* indicator specifies *extended* mode, the *base locator* addresses the last byte of the base string, and that byte value is not a shift out, hex 0E character. In this case, complete verification of the extended character and relation comparison could not be performed. If this extended character is to be processed, it must be done through another execution of this instruction where both bytes of the character can be input to the instruction within the confines of the base string.

Completion code values:

•

- (-1) Low: A character value was found in the base string which satisfies the criteria specified in the controls and options operands in that the comparison character is of lower string value to the base string character.

- (0) Equal: A character value was found in the base string which satisfies the criteria specified in the controls and options operands in that the comparison character is of equal string value to the base string character.
- (1) High: A character value was found in the base string which satisfies the criteria specified in the controls and options operands in that the comparison character is of higher string value to the base string character.
- (2) Not found: A character value was not found in the base string which satisfied the criteria specified in the controls and options operands.
- (3) Escape code encountered: The *test for escape code* option was specified and a character with a value less than hex 40 was found which is not a valid mode control character for the type of scan requested. That is, hex 0E and hex 0F are valid mode control characters while the scan is being performed in extended mode, but are not when the scan is being performed in simple mode.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

32 Scalar Specification

3203 Scalar Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Scan with Control (SCANWC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-8]
SCANWC 10E4		Base locator	Controls	Options	Escape target or null	
SCANWCB 1CE4	Branch options	Base locator	Controls	Options	Escape target or null	Branch targets
SCANWCI 18E4	Indicator options	Base locator	Controls	Options	Escape target or null	Indicator targets

Operand 1: Space pointer.

Operand 2: Character(8) variable scalar.

Operand 3: Character(4) constant scalar.

Operand 4: Instruction number, relative instruction number, branch point, instruction pointer, instruction definition list element, or null.

Operand 5-8:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The base string to be scanned is specified by the *base locator* and *controls* operands. The *base locator* addresses the first character of the base string. The *controls* specifies the length of the base string in the base length field.

The scan operation begins at the left end of the base string and continues character by character, left-to-right. The scan operation can be performed on a base string which contains all simple (1-byte) or all extended (2-byte) character codes or a mixture of the two. When the base string is being interpreted in simple character mode, the operation moves through the base string one byte at a time. When the base string is being interpreted in extended character mode, the operation moves through the base string 2 bytes at a time. The character string value of the *base locator* operand is scanned for occurrences of a character value satisfying the criteria specified in the *controls* and *options* operands.

The scan is completed by updating the *base* locator and *controls* operands with scan status when a character value being scanned for is found, the end of the base string is encountered, or an escape code is encountered when the *escape target* operand is specified. The *base locator* is set with addressability to the character (simple or extended) which caused the instruction to complete execution. The *controls* operand is set with information which identifies the mode (simple or extended) of the base string character addressed by the *base locator* and which provides for resumption of the scan operation with minimal overhead.

The *controls* and *options* operands specify the modes to be used in interpreting characters during the scan operation. Characters can be interpreted in one of two character modes: simple (1-byte) and extended

(2-byte). Additionally, the base string can be scanned in one of two scan modes, mixed (base string may contain a mixture of both character modes) and nonmixed (base string contains one mode of characters).

When the mixed scan mode is specified in the options operand, the base string is interpreted as containing a mixture of simple and extended character codes. The mode, simple or extended, with which the string is to be interpreted, is controlled initially by the base mode indicator in the *controls* operand and thereafter by mode control characters imbedded in the base string. The mode control characters are as follows:

Hex 0E =	Shift out (SO) of simple character mode to extended mode.
Hex 0F =	Shift in (SI) to simple character mode from extended mode. This is only recognized if it occurs in the first byte position of an extended character code.

When the nonmixed scan mode is specified in the *options* operand, the base string is interpreted using only the character mode specified by the base mode indicator in the *controls* operand. Character mode shifting can not occur because no mode control characters are recognized when scanning in nonmixed mode.

The *base locator* operand is a space pointer which is both input to and output from the instruction. On input, it locates the first character of the base string to be processed. On output, it locates the character of the base string which caused the instruction to complete.

The *controls* operand must be a character scalar which specifies additional information to be used to control the scan operation. It must be at least 8 bytes long and have the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Controls operand	Char(8)	
0	0		Control indicators	Char(1)
0	0		Base mode	Bit 0
			0 = Simple	
			1 = Extended	
0	0		Comparison character mode	Bit 1
			0 = Simple	
			1 = Extended	
0	0		Reserved (must be 0)	Bits 2-6
0	0		Scan state	Bit 7
			0 = Resume scan	
			1 = Start scan	
1	1		Reserved	Char(1)
2	2		Comparison characters	Char(2)
4	4		Reserved	Char(1)
5	5		Base end	Char(3)
5	5		Instruction work area	Char(1)
6	6		Base length	Char(2)
8	8	—- End —-		

Only the first 8 bytes of the *controls* operand are used. Any excess bytes are ignored. Reserved fields must contain binary 0s.

The **base mode** is both input to and output from the instruction. In either case, it specifies the mode of the character in the base string currently addressed by the *base locator*.

The **comparison character mode** is not changed by the instruction. It specifies the mode of the comparison character contained in the *controls* operand.

The **scan state** is both input to and output from the instruction. As input, it indicates whether the scan operation for the base string is being started or resumed. If it is being *started*, the instruction assumes that the *base length* value in the *base end* field of the *controls* operand specifies the length of the base string, and the instruction work area value is ignored. If it is being *resumed*, the instruction assumes the *base end* field has been set by a prior start scan execution of the instruction with an internal machine value identifying the end of the base string.

For a *start scan* execution of the instruction, the *scan state* field is reset to indicate resume scan to provide for subsequent resumption of the scan operation. Additionally, for a *start scan* execution of the instruction, the *base end* field is set with an internally optimized value which identifies the end of the base string being scanned. This value then overlays the values which were in the instruction work area and base length fields on input to the instruction. Predictable operation of the instruction on a *resume scan* execution depends upon this base end field being left intact with the value set by the start scan execution.

For a *resume scan* execution of the instruction, the *scan state* and *base end* fields are unchanged. For a *resume scan* execution of the instruction, *base locator* must not contain a teraspace address or an *unsupported space use* (hex 0607) exception is signaled.

The **comparison character** is input to the instruction. It specifies a character code to be used in the comparisons performed during the scanning of the base string. The *comparison character mode* in the control indicators specifies the mode (simple or extended) of the comparison character. If it is a *simple* character, the first byte of the *comparison character* field is ignored and the *comparison character* is assumed to be specified in the second byte. If it is an *extended* character, the *comparison character* is specified as a 2-byte value in the *comparison character* field.

The **base end** field is both input to and output from the instruction. It contains data which identifies the end of the base string. Initially, for a start scan execution of the instruction, it contains the length of the base string in the **base length** field. Additionally, the *base end* field is used to retain information over multiple instruction executions which provides for minimizing the overhead required to resume the scan operation for a particular base string. This information is set on the initial *start scan* execution of the instruction and is used during subsequent *resume scan* executions of the instruction to determine the end of the base string to be scanned. If the end of the base string being scanned must be altered during iterative usage of this instruction, a *start scan* execution of the instruction must be performed to provide for correctly resetting the internally optimized value to be stored in the *base end* from the values specified in the *base locator* operand and *base length* field.

For the special case of a *start scan* execution where a length value of zero (no characters to scan) is specified in the *base length* field, the instruction results in a *not found* resultant condition. In this case, the base string is not verified and the *scan state* indicator, the *base end* field, and the *base locator* are not changed.

The *options* operand must be a character scalar which specifies the options to be used to control the scan operation. It must be at least 4 bytes in length and has the following format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Options operand	Char(4)	
0	0		Options indicators	Char(1)
1	1		Reserved	Char(3)
4	4	—- End —-		

The *options* operand must be specified as a constant character scalar.

Only the first 4 bytes of the options operand are used. Any excess bytes are ignored. Reserved fields must contain binary 0s. The **option indicators** field has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Option indicators	Char(1)	
0	0		Reserved	Bit 0
0	0		Scan mode	Bit 1
			0 = Mixed	
			1 = Nonmixed	
0	0		Reserved	Bits 2-3
0	0		Comparison relation	Bits 4-6
0	0		Equal, (=) relation	Bit 4
0	0		Less than, (<) relation	Bit 5
0	0		Greater than, (>) relation	Bit 6
			0 = No match on relation	
			1 = Match on relation	
0	0		Reserved	Bit 7
1	1	—- End —-		

The **scan mode** specifies whether the base string contains a mixture of character modes, or contains all one mode of characters; that is, whether or not mode control characters should be recognized in the base string. *Mixed* specifies that there is a mixture of character modes and, therefore, mode control characters should be recognized. *Nonmixed* specifies that there is not a mixture of character modes and, therefore, mode control characters should not be recognized. Note that the *base mode* indicator in the controls operand specifies the character mode of the base string character addressed by the *base locator*.

The **comparison relation** specifies the relation or relations of the comparison character to characters of the base string which will satisfy the scan operation and cause completion of the instruction with one of the *high, low, or equal* resultant conditions. Multiple relations may be specified in conjunction with one another. Specifying all relations insures a match against any character in the base string which is of the same mode as the comparison character. Specifying no relation insures a *not found* resultant condition, in the absence of an escape due to verification, regardless of the values of the characters in the base string which match the mode of the comparison character.

An example of comparison scanning is a scan of simple mode characters for a value less than hex 40. This could be done by specifying a *comparison character* of hex 40 and a *comparison relation* of *greater* than in conjunction with a branch option for the resultant condition of *high*. This could also be done by specifying a *comparison character* of hex 3F and *comparison relations* of *equal* and *greater* than in conjunction with branch options for *equal* and *high*. The target of the branch options in either case would be the instructions to process the character less than hex 40 in value.

The *escape target* operand controls the verification of bytes of the base string for values less than hex 40. Verification, if requested, is always performed in conjunction with whatever comparison processing has been requested. That is, verification is performed even if no comparison relation is specified. This operand is discussed in more detail in the following material.

During the scan operation, the characters of the base string which are not of the same mode as the comparison character are skipped over until the mode of the characters being processed is the same as the mode of the comparison character. The operation then proceeds by comparing the comparison character with each of the characters of the base string. These comparisons behave as if the characters were being compared in the Compare Bytes Left Adjusted (CMPBLA) instruction.

If a base string character satisfying the criteria specified in the *controls* and *options* operands is found, the *base locator* is set to address the first byte of it, the *base mode* indicator is set to indicate the mode of the base string as of that character, and the instruction is completed with the appropriate resultant condition based on the *comparison relation* (*high*, *low*, or *equal*) of the *comparison character* to the base string character.

If a matching base string character is not found prior to encountering a mode change, the characters of the base string are again skipped over until the mode of the characters being processed is the same as the mode of the comparison character before comparisons are resumed.

If a matching base string character is not found prior to encountering the end of the base string, the base location is set to address the first byte of the character encountered at the end of the base string, the *base mode* indicator is set to indicate the mode of the base string as of that character, and the instruction is completed with the *not found* resultant condition. A mode control string results in the changing of the base string mode, but the base locator is left addressing the mode control character.

If the *escape target* operand is specified (operand 4 is not null), verifications are performed on the characters of the base string prior to their being skipped or compared with the comparison character. Each byte of the base string is checked for a value less than hex 40. Additionally, for a *mixed* scan mode, when such a value is encountered, it is then determined if it is a valid mode control character.

•

- Hex 0E (S0) when the base string is being interpreted in simple character mode.
- Hex 0F (SI) in the left byte of the character code when the base string is being interpreted in extended character mode.

If a byte value of less than hex 40 is not a valid mode control character, it is considered to be an escape code. The *base locator* is set to address the first byte of the base string character (simple or extended) which contains the escape code, the *base mode* indicator is set to indicate the mode of the base string as of that character, and a branch is taken to the target specified by the *escape target* operand. When the escape target branch is performed, the value of any optional indicator operands is meaningless.

If the *escape target* operand is not specified (operand 4 is null), verifications of the character codes in the base string are not performed. However, for a *mixed* scan mode, mode control values are always processed as described previously under the discussion of the mixed scan mode.

If possible, use a space pointer machine object for the base locator, operand 1. Appreciably less overhead is incurred in accessing and storing the value of the *base locator* if this is done.

If possible, specify through its ODT definition, the *controls* operand on an 8-byte multiple (doubleword) boundary relative to the start of the space containing it. Appreciably less overhead is incurred in accessing and storing the value of the *controls* if this is done.

For the case where a base string is to be just scanned for byte values less than hex 40, two techniques can be used.

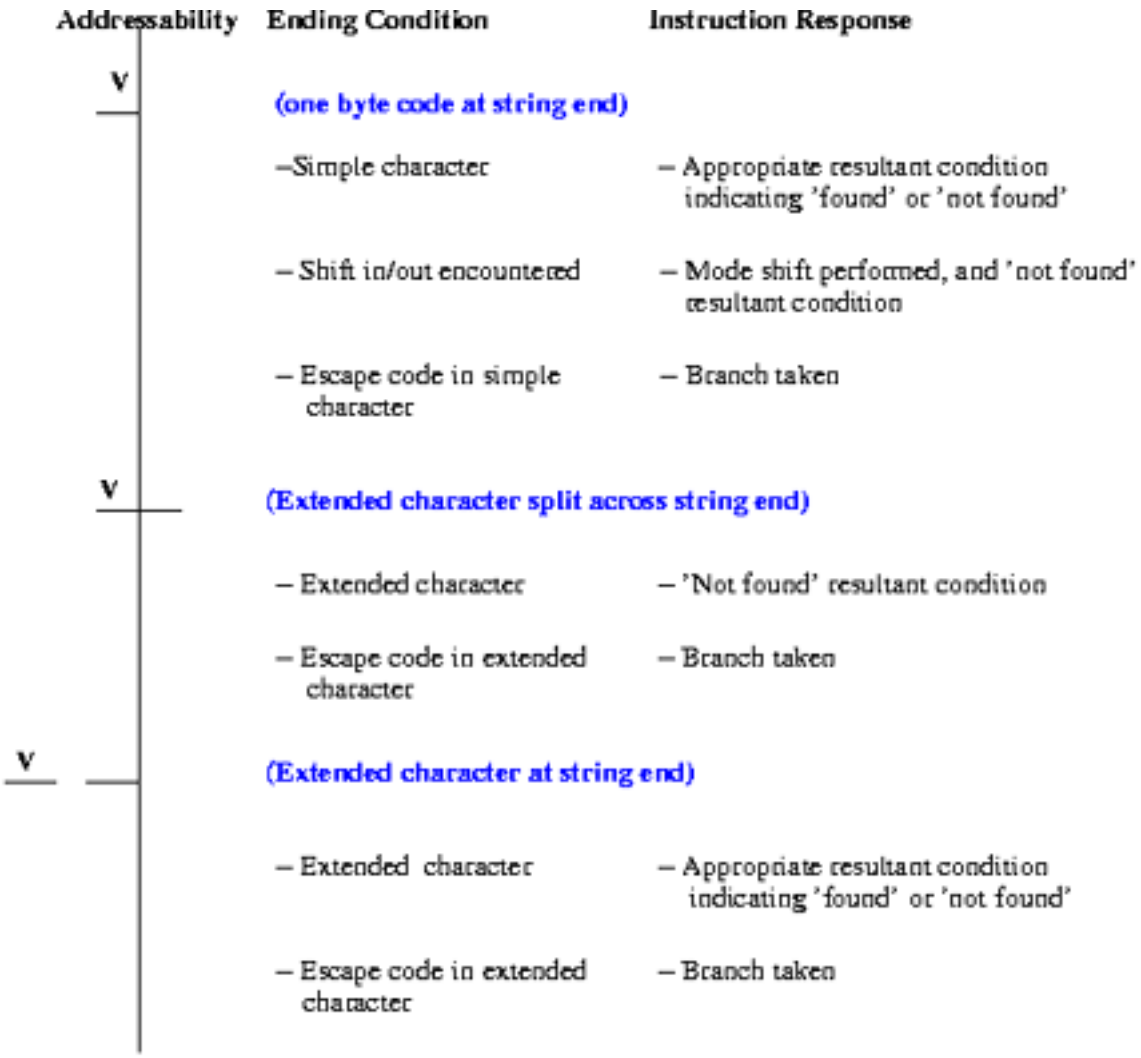
•

- A direct simple mode scan for a value less than hex 40 without usage of the *escape target* verification feature.
- A scan for any character with usage of the *escape target* verification feature.

  The direct scan approach, the former, is the more efficient.

The following diagram defines the various conditions which can be encountered at the end of the base string and what the base locator addressability is in each case. The solid vertical line represents the end of the base string. The dashes represent the bytes before and after the base string end. The V is positioned over the byte addressed by the base locator in each case. These are the conditions which can be encountered when the *base locator* input to the instruction addresses a byte prior to the base string end.

When the base length field specifies a value of zero for a start scan execution of the instruction, or the input base locator addresses a point beyond the end of the instruction, no processing is performed and the instruction is immediately completed with the *not found* resultant condition.



An analysis of the diagram shows that normally, after appropriate processing for the particular *found, not found, or escape* condition, the scan can be restarted at the byte of data which would follow the base string end in the data stream being scanned. Any mode shift required by an ending mode control character will have been performed.

However, one ending condition may require subsequent resumption of the scan at the character encountered at the end of the base string. This is the case where the instruction completes with the *not found* resultant condition and the base string ends with an extended character split across string end. That is, the *base mode* indicator specifies *extended* mode, the *base locator* addresses the last byte of the base string, and that byte value is not a shift out, hex 0E character. In this case, complete verification of the extended character and relation comparison could not be performed. If this extended character is to be processed, it must be done through another execution of the Scan instruction where both bytes of the character can be input to the instruction within the confines of the base string.

Resultant Conditions:

•

- Equal: A character value was found in the base string which satisfies the criteria specified in the *controls* and *options* operands in that the comparison character is of equal string value to the base string character.
- High: A character value was found in the base string which satisfies the criteria specified in the *controls* and *options* operands in that the comparison character is of higher string value to the base string character.
- Low: A character value was found in the base string which satisfies the criteria specified in the *controls* and *options* operands in that the comparison character is of lower string value to the base string character.
- Not found: A character value was not found in the base string which satisfied the criteria specified in the *controls* and *options* operands.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Search (SEARCH)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-6]
SEARCH 1084		Receiver	Array	Find	Location	
SEARCHB 1C84	Branch options	Receiver	Array	Find	Location	Branch targets
SEARCHI 1884	Indicator options	Receiver	Array	Find	Location	Indicator targets

Operand 1: Binary variable scalar or binary variable array.

Operand 2: Character array or numeric array.

Operand 3: Character variable scalar or numeric variable scalar.

Operand 4: Binary scalar.

Operand 5-6:

•
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The portions of the *array* operand indicated by the *location* operand are searched for occurrences of the value indicated in the *find* operand.

The operation begins with the first element of the *array* operand and continues element by element, comparing those characters of each element (beginning with the character indicated in the location operand) with the characters of the *find* operand. The *location* operand contains an integer value representing the relative location of the first character in each element to be used to begin the compare.

The integer value of the *location* operand must range from 1 to L, where L is the length of the *array* operand elements; otherwise, a *scalar value invalid* (hex 3203) exception is signaled. A value of 1 indicates the leftmost character of each element.

The length of the *find* operand must not be so large that it exceeds the length of the *array* operand elements when used with the *location* operand value; otherwise, a *length conformance* (hex 0C08) exception is signaled. The array element length used is the length of the array scalar elements and not the length of the entire array element, which can be larger in noncontiguous arrays.

The *array* and *find* operands can be either character or numeric. Any numeric operands are interpreted as logical character strings. The compares between these operands are performed at the length of the *find* operand and function as if they were being compared in the Compare Bytes Left-Adjusted (CMPBLA) instruction.

As each occurrence of the *find* value is encountered, the integer value of the index for this array element is placed in the *receiver* operand. If the *receiver* operand is a scalar, only the first element containing the *find* value is noted. If the *receiver* operand is an array, as many occurrences as there are elements within the *receiver* array are noted.

If the value of the index for an array element containing an occurrence of the *find* value is too large to be contained in the *receiver*, a *size* (hex 0C0A) exception is signaled.

The operation continues until no more occurrences of elements containing the *find* value can be noted in the *receiver* operand or until the *array* operand has been completely searched. When the second condition occurs, the *receiver* value is set to LB-1, where LB is the value of the lower bound index of the array. If LB is the most negative 32-bit integer, then LB-1 is the most positive 32-bit integer; otherwise, LB-1 is 1 less than LB. If the *receiver* operand is an array, all its remaining elements are also set to LB-1. The *find* operand can be a variable length substring compound operand.

Resultant Conditions: The numeric value(s) of the *receiver* operand is either LB-1 or in the range LB through UB, where UB is the value of the upper bound index of the array. When the *receiver* is LB-1, the resultant condition is *zero*. When the *receiver* is in the range LB through UB, the resultant condition is *positive*. When the *receiver* is an array, the resultant condition is *zero* if all elements are LB-1; otherwise, it is *positive*. The resultant condition is unpredictable when the *no binary size exception* program template option is used.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C08 Length Conformance

0C0A Size

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Sense Exception Description (SNSEXCPD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03E3	Attribute receiver	Invocation template	Exception template

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Space pointer.

Note: A change has been made in the way in which exceptions are handled for bound programs. This instruction is intended for use with non-bound programs, but can be used against bound programs. The data that is returned when a bound program is accessed will always say that there is an external handler for the sensed exception, that there is no exception data being returned and a starting exception description number of 0.

Description: This instruction searches the invocation specified by operand 2 for an exception description that matches the *exception identifier* and *compare value* specified by operand 3 and returns the

user data and *exception handling action* specified in the exception description. The exception descriptions of the invocation are searched in ascending Object Definition Table (ODT) number sequence.

The template identified by operand 1 must be 16-byte aligned.

The format of the *attribute receiver* is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Template size	Char(8)		
0	0		Number of bytes provided for materialization		Bin(4)
4	4		Number of bytes available for materialization		Bin(4)
8	8	Control flags	Char(2)		
8	8		Exception handling action		Bits 0-2
			000 =	Do not handle- Ignore occurrence of exception and continue processing	
			010 =	Do not handle- Continue search for an exception description by resignaling the exception to the immediately preceding invocation	
			100 =	Defer handling- Save exception data for later exception handling	
			101 =	Pass control to the specified exception handler	
8	8		No data		Bit 3
			0 =	Exception data is returned	
			1 =	Exception data is not returned	
8	8		Reserved (binary 0)		Bit 4
8	8		User data indicator		Bit 5
			0 =	User data not present	
			1 =	User data present	
8	8		Reserved (binary 0)		Bits 6-7
8	8		Exception handler type		Bits 8-9
			00 =	External entry point	
			01 =	Internal entry point	
			10 =	Branch point	
8	8		Reserved (binary 0)		Bits 10-15
10	A	Relative exception description number	Bin(2)		
12	C	Reserved (binary 0)	Char(4)		
16	10	Pointer to user data (binary 0 if value of user data indicator is 0)	Space pointer		
32	20	—- End —-			

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exception is signaled in the event the receiver contains insufficient area for the materialization, other than the *materialization length invalid* (hex 3803) exception described previously.

The **relative exception description number** field identifies the relative number of the exception description that matched the search criteria. The order of definition of the exception descriptions in the ODT determines the value of the index. A value of 1 indicates that the first exception description defined in the ODT matched the search criteria.

The format of the *invocation template* is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Invocation address/offset	Space pointer or Invocation pointer	
16	10	Search flags	Char(2)	
16	10		Use offset option	Bit 0
			0 = Use invocation address as a pointer value	
			1 = Use invocation address as an offset value	
16	10		Reserved (binary 0)	Bits 1-15
18	12	First exception description to search	Bin(2)	
20	14	— End —		

The template identified by operand 2 must be 16-byte aligned. The **invocation address/offset** field is a space/invocation pointer that identifies the invocation to be searched. The invocation is searched for a matching exception description. If the *invocation address* locates either an invalid invocation or the invocation stack base entry, the *invalid invocation address* (hex 1603) exception is signaled.

The *invocation address/offset* field can also be an offset value from the current requesting invocation to the invocation to be searched. This is setting the **use offset option** bit field that follows the *invocation address* field to 1. If the *invocation offset* value locates the invocation stack base entry, the *invocation offset outside range of current stack* (hex 2C1A) exception is signaled. If the *invocation offset* value is positive or zero, a *materialization length invalid* (hex 3803) exception is signaled.

The **first exception description to search** field specifies the relative number of the exception description to be used to start the search. The number must be a nonzero positive binary number determined by the order of definition of exception descriptions in the ODT. A value of 1 indicates that the first exception description in the invocation is to be used to begin the search. If the value is greater than the number of exception descriptions for the invocation, the operand 1 template is materialized with the *number of bytes available* for materialization set to 0.

The operand 3 *exception template* specifies the exception-related data to be used as a search argument. The format of the template is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size	Char(8)	
0	0		Number of bytes provided for materialization (must be at least 44)	Bin(4)
4	4		Number of bytes available for materialization	Bin(4) +
8	8	Exception identifier	Char(2)	
10	A	Compare value length (maximum of 32)	Bin(2)	
12	C	Compare value	Char(32)	
44	2C	—- End —-		

Note: Fields noted with a plus sign (+) are ignored by the instruction.

The **exception identifier** in the exception description can be specified in one of the following ways:

Hex 0000 =	Any exception ID will result in a match
Hex nn00 =	Any exception ID in class nn will result in a match
Hex nnmm =	Only exception ID nnmm will result in a match

If a match on *exception ID* is detected, the corresponding **compare values** are matched. If the *compare value length* in the exception description is less than the *compare value* in the search template, the length of the *compare value* in the exception description is used for the match. If the *compare value length* in the exception description is greater than the *compare value* in the search template, an automatic mismatch results.

If a match on *exception ID* and *compare value* is detected, the exception handling action of the exception description determines which of the following actions is taken:

IGNORE	The operand 1 template is materialized.
DISABLE	The exception description is bypassed and the search for an exception description continues with the next exception description defined for the invocation.
RESIGNAL	The operand 1 template is materialized.
DEFER	The operand 1 template is materialized.
HANDLE	The operand 1 template is materialized.

If no exception description of the invocation matches the *exception ID* and *compare value* of operand 3, the *number of bytes available for materialization* on the operand 1 template is set to 0.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

16 Exception Management

1603 Invalid Invocation Address

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C1A Invocation Offset Outside Range of Current Stack

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3802 Template Size Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Set Access State (SETACST)

Op Code (Hex)	Operand 1
0341	Access state template

Operand 1: Space pointer.

Bound program access

```
Built-in number for SETACST is 70.
SETACST (
         access_state_template   : address
)
```

Description: The instruction specifies the access state (which specifies the desired speed of access) that the issuing thread has for a set of objects or subobject elements in the execution interval following the execution of the instruction. The specification of an access state for an object momentarily preempts the machine's normal management of an object.

Note: This instruction should be used with caution when the *pointer to object whose access state is to be changed* field in the template below points to a process space (i.e. static storage, automatic storage, and activation group-based heap space storage). These process spaces may be shared by other programs, so explicit access management may affect those other programs. This instruction should be used with caution when the *pointer to object whose access state is to be changed* field in the template below points to handle-based heap space storage. Handle-based heap space storage may be shared by other threads in the process, so explicit access management may affect programs in those other threads.

CAUTION:

MI system objects can be implemented with one or more storage structures for the functional part (encapsulated part) of the object. Unless explicitly noted otherwise, if a system pointer is specified for *pointer to object whose access state is to be changed*, the SETACST operations only act on the first (base) storage structure of an MI object's functional part and the object's primary associated space. The MI objects that are implemented with multiple storage structures are:

- Cursor
- Byte stream file
- Byte string space
- Data space
- Data space index
- Dump space
- Independent index
- Journal space
- Module
- Program
- Queue
- Queue space
- User profile

The *access state template* must be aligned on a 16-byte boundary. The format is:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Number of objects to be acted upon	Bin(4)
4	4	Reserved (binary 0)	Char(12)
16	10	Access state specifications (repeated as many times as necessary)	[*] Char(32)
16	10		Pointer to object whose access state is to be changed
32	20		Access state code
33	21		Reserved (binary 0)
36	24		Access state parameter
36	24		Access pool ID
40	28		Space length
44	2C		Operational object size
			Note: This value is returned for some of the *access stat*
*	*	—- End —-	

The **number of objects** field specifies how many objects are potential candidates for access state modification. An **access state specification** is included for each object to be acted upon.

The **pointer to object** field identifies the object or space which is to be acted upon. For the space associated with a system object, the space pointer may address any byte in the space. This pointer is followed by parameters that define in detail the action to be applied to the object.

The **access state code** designates the desired access state. The allowed values are as follows:

Access State Code (Hex)	Function and Required Parameter
00	No operations are performed.

Access State Code (Hex)	Function and Required Parameter
01	Associated object is moved into main storage (if not already there) synchronously with the execution of the instruction.
02	Associated object is moved into main storage (if not already there) asynchronously with the execution of the instruction. This operation will be applied to all internal storage areas for queue objects.
03	Associated object is placed in main storage without regard to the current contents of the object. This causes access to secondary storage to be reduced or eliminated. For this access state code, a space pointer must be provided.
04	Associated object is removed from main storage in a manner which reduces or eliminates access to secondary storage. Content of the object is unpredictable after this operation. For this access state code, a space pointer must be provided.
10	The object is synchronously ensured (changes written to auxiliary storage) and then removed from main storage. If the system pointer points to an object whose type is not a cursor, data space, data space index, program, or space then an *object not eligible for operation* (hex 2204) exception is signaled. This option returns a number in the *operational object size* field. The unit assumed is the machine minimum transfer size (page size). The value returned is the total size of the operational parts of the object examined/processed, including the associated space (if there is one). **Note:** This number is not the number of pages written or removed, but rather, is the total size of the object being processed. Some, all or none of the object may be in mainstore prior to the execution of the instruction. The *space length* field must be zero for this operation. The entire associated space, if any, will be processed with the rest of the object's storage. The *access pool ID* field is ignored for this operation. The associated pointer to the object must be a system pointer. This operation will be applied to all internal storage areas for queue objects.
18	This operation essentially combines the functions of a 10 code followed by asynchronously bringing the operational parts of the object into main storage. The object is brought into the main storage pool identified by the *access pool ID* field. If the system pointer points to an object whose type is not a cursor, data space, data space index, program, or space then an *object not eligible for operation* (hex 2204) exception is signaled. **Note:** Because this function first removes the object from main storage and then brings it into main storage, this can be used to "move" an object from one main storage pool to another. This option returns a number in the *operational object size* field. The unit assumed is the machine minimum transfer size (page size). The value returned is the total size of the object processed. **Note:** If this value is larger than the size of the main storage pool being used, unpredictable parts of the object will be resident in the main storage pool following processing. A preceding access code of 40 is ignored for this operation. The *space length* field must be zero for this operation. The entire associated space, if any, will be processed with the rest of the object's storage. The *access pool ID* field must be specified for this access code. It must be one of the storage pools existing in the machine as defined by the machine attribute. The associated pointer to the object must be a system pointer. This operation will be applied to all internal storage areas for cursor, data space and data space index objects.
20	Associated object attributes are moved into main storage synchronous with the instruction's execution. The associated attributes are the attributes that are common to all system objects. The associated pointer to object must be a resolved system pointer. The "space length" field is ignored for this access code.
21	Associated object attributes are moved into main storage asynchronous with the instruction's execution. The associated attributes are the attributes that are common to all system objects. The associated pointer to object must be a resolved system pointer. The "space length" field is ignored for this access code.

Access State Code (Hex)	Function and Required Parameter
22	Common associated object attributes plus some specified amount of object-specific attributes are moved into main storage synchronous with the instruction's execution. The common associated attributes are the attributes that are common to all system objects. The object-specific attributes are attributes that vary from one object type to another. The amount of these attributes brought into main storage is controlled by the *space length* field. **Note:** This use of *space length* is not consistent with the name of the field. For this code, the *space length* field does not control the size of any associated space processing, it controls the length of object-specific attributes processed. The *space length* field works in the following manner: it specifies the amount of storage above and beyond the common object attributes which will be synchronously brought into storage. Therefore, a *space length* of 0 is valid, and results in an operation identical to access code 20. The associated pointer to object must be a resolved system pointer.
23	Common associated object attributes plus some specified amount of object-specific attributes are moved into main storage asynchronous with the instruction's execution. The common associated attributes are the attributes that are common to all system objects. The object-specific attributes are attributes that vary from one object type to another. The amount of these attributes brought into main storage is controlled by the *space length* field. **Note:** This use of *space length* is not consistent with the name of the field. For this code, the *space length* field does not control the size of any associated space processing, it controls the length of object-specific attributes processed. The *space length* field works in the following manner: it specifies the amount of storage above and beyond the common object attributes which will be asynchronously brought into storage. Therefore, a *space length* of 0 is valid, and results in an operation identical to access code 21. The associated pointer to object must be a resolved system pointer.
30	The associated space of the object is moved into main storage (if not already there) synchronously with the execution of the instruction. The *space length* field is honored for this operation. The associated pointer to the object must be a system pointer.
31	The associated space of the object is moved into main storage (if not already there) asynchronously with the execution of the instruction. The *space length* field is honored for this operation. The associated pointer to the object must be a system pointer.
40	Perform no operation on the associated object. The main storage occupied by this object is to be used, if possible, to satisfy the request in the next access state specification entry. Either a space or system pointer may be provided for this access state code. This operation will be applied to all internal storage areas for queue objects.
41	Wait for any previously issued but incomplete hex 81 or hex 91 access state code operations to complete. This includes all previous hex 81 and hex 91 operations that may have been performed on previous Set Access State instructions within the current thread as well as those that may have been issued in previous access state specification entries in the current instruction. The pointer is ignored for this access state code entry. This operation will be applied to all internal storage areas for queue objects.
80	Object should be written and it is not needed in main storage by issuing thread. Object is written to nonvolatile storage synchronously with the execution of the instruction. Any main storage that the object occupied is then marked as to make it quickly available for replacement. This operation will be applied to all internal storage areas for queue objects.
81	Object should be written and it is not needed in main storage by issuing process. Object is written to nonvolatile storage asynchronously with the execution of the instruction. Any main storage that the object occupied is then marked as to make it quickly available for replacement. If desired, the thread can synchronize with any outstanding hex 81 access state operation by issuing a hex 41 access state operation either within the current instruction or during a subsequent Set Access State instruction. This operation will be applied to all internal storage areas for queue objects.
90	Associated object must be insured, but is still needed in main storage. Object is written to nonvolatile storage synchronously with the execution of the instruction. Unlike access state codes hex 80 and hex 81, this access state code does not mark any main storage occupied by the object as to make it quickly available for replacement. This operation will be applied to all internal storage areas for queue objects.

Access State Code (Hex)	Function and Required Parameter
91	Associated object must be insured, but is still needed in main storage. Object is written to nonvolatile storage asynchronously with the execution of the instruction. Unlike access state codes hex 80 and hex 81, this access state code does not mark any main storage occupied by the object as to make it quickly available for replacement. If desired, the thread can synchronize with any outstanding hex 91 access state operation by issuing a hex 41 access state operation either within the current instruction or during a subsequent Set Access State instruction.

Access state codes hex 03 and hex 04 may be used for spaces only. The pointer to the object in the access state specification must be a space pointer. Otherwise, the *pointer type invalid* (hex 2402) exception is signaled.

Access state code hex 40 may be used in conjunction with access state codes hex 01, hex 02, or hex 03. The access state specification entry with access state code hex 40 must immediately precede the access state specification entry with access state code hex 01, hex 02, or hex 03 with which it is to be combined. The pointer to the object in both entries must be a space pointer. Otherwise, the *pointer type invalid* (hex 2402) exception is signaled. The *access state parameter* field in the *access state specification entry* with code hex 40 is ignored. The *access pool ID* and the *space length* in the entry with access state code hex 01, hex 02, or hex 03 are used.

The **access pool ID** field indicates the desired main storage pool in which the object is to be placed (0-16). The storage pool ID entry is treated as a 4-byte logical binary value. When a 0 storage pool ID is specified, the storage pool associated with the issuing thread is used.

The **space length** field designates the part of the space associated with the object to be operated on. If the pointer to the object entry is a system pointer, the operation begins with the first byte of the space. If the pointer to the object entry is a space pointer that specifies a location, the operation proceeds for the number of storage units that are designated. No exception is signaled when the number of referenced bytes of the space are not allocated. When operations on objects are designated by system pointers, this operation is performed in addition to the access state modification of the object. This entry is ignored for *access state codes* hex 20 and hex 21. This entry will be truncated to a maximum of 65,536 for *access state codes* immediately following access state code 40.

The **operational object size** field is a value which is ignored upon input to the instruction and is set by the instruction for access codes 10 and 18. It represents, in units of minimum machine transfer size, the total size of the object which could/did participate in the operation. The parts of an object which are considered ″operational″ are decided by the machine and does include the associated space, if any.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

  –

  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize

  –

  – Contexts referenced for address resolution

Exceptions

04 Access State

- 0401 Access State Specification Invalid

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

0A Authorization

- 0A01 Unauthorized for Operation

10 Damage Encountered

- 1004 System Object Damage State
- 1005 Authority Verification Terminated Due to Damaged Object
- 1044 Partial System Object Damage

1A Lock State

- 1A01 Invalid Lock State

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2204 Object Not Eligible for Operation
- 2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2404 Pointer Not Resolved

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Set Activation Group-Based Heap Space Storage Mark (SETHSSMK)

Op Code (Hex)	Operand 1	Operand 2
03B6	Mark identifier	Heap identifier

Operand 1: Space pointer data object.

Operand 2: Binary(4) scalar.

Bound program access

```
Built-in number for SETHSSMK is 118.
SETHSSMK (
          mark_identifier   : address of space pointer(16)
          heap_identifier   : address of signed binary(4)
)
```

Note: The term ″heap space″ in this instruction refers to an ″activation group-based heap space″.

Description: The heap space identified by operand 2 is marked and the *mark identifier* is returned in operand 1.

Marking a heap space allows a subsequent Free Activation Group-Based Heap Space Storage from Mark (FREHSSMK) instruction, using the *mark identifier* returned in operand 1, to free all outstanding

allocations that were performed against the heap space since the heap space was marked with that mark identifier. This relieves the user of performing a Free Activation Group-Based Heap Space Storage (FREHSS) for every individual heap space allocation.

A heap space may have multiple marks.

The *heap identifier* specified in operand 2 is the identifier that was returned on the Create Activation Group-Based Heap Space (CRTHS) instruction. An attempt to mark the default heap space (heap identifier value of 0) will result in an *invalid request* (hex 4502) exception. An attempt to mark a heap space that has been created to not allow a Set Heap Space Storage Mark will result in an *invalid request* (hex 4502) exception.

Operand 2 is not modified by the instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C04 Object Storage Limit Exceeded

1C09 Auxiliary Storage Pool Number Invalid

20 Machine Support

2002 Machine Check

2003 Function Check

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

45 Heap Space

4501 Invalid Heap Identifier

4502 Invalid Request

4503 Heap Space Full

4505 Heap Space Destroyed

4506 Invalid Heap Space Condition

Set Argument List Length (SETALLEN)

Op Code (Hex)	Operand 1	Operand 2
0242	Argument list	Length

Operand 1: Operand list.

Operand 2: Binary scalar.

Description: This instruction specifies the number of arguments to be passed on a succeeding Call External or Transfer Control instruction. The current length of the variable-length operand list (used as an argument list) specified by operand 1 is modified to the value indicated in the binary scalar specified by operand 2. This length value specifies the number of arguments (starting from the first) to be passed from the list when the operand list is referenced on a Call External or Transfer Control instruction.

Only variable-length operand lists with the argument list attribute may be modified by the instruction.

The value in operand 2 may range from 0 (meaning no arguments are to be passed) to the maximum size specified in the ODT definition of the operand list (meaning all defined arguments are to be passed).

The length of the argument list remains in effect for the duration of the current invocation or until a Set Argument List Length instruction is issued against this operand list.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0803 Argument List Length Modification Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Set Bit in String (SETBTS)

Op Code (Hex)	Operand 1	Operand 2
101E	Receiver	Offset

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Binary scalar.

Bound program access

```
Built-in number for SETBTS is 3.
SETBTS (
        receiver   : address
        offset     : unsigned binary(4)
)
```

The *offset* operand must be between 0 and 65,535.

Description: Sets the bit of the *receiver* operand as indicated by the bit *offset* operand.

The selected bit from the *receiver* operand is set to a value of binary 1.

The *receiver* operand can be a character or numeric variable. The leftmost bytes of the *receiver* operand are used in the operation. The *receiver* operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus 1.

The *offset* operand indicates which bit of the *receiver* operand is to be set, with an offset of zero indicating the leftmost bit of the leftmost byte of the *receiver* operand. This value may be specified as a constant or any valid binary scalar variable.

If a offset value less than zero or beyond the length of the *receiver* is specified a *scalar value invalid* (hex 3203) exception is signaled.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Set Computational Attributes (SETCA)

Bound program access

```
Built-in number for SETCA is 146.
SETCA (
       new_attributes   : unsigned binary(4) value which contains the new
                          computational attribute values
       selector         : unsigned binary(4) literal value; its rightmost
                          byte specifies the computational attributes to
                          set
)
```

Description: The right-most byte of *selector* specifies the computational attributes to modify. The format of this byte is as follows:

Bit	Definition
0-3	Reserved (must be 0)
4	Exception mask
5	Reserved (must be 0)
6	Exception occurrence
7	Rounding mode

All other bytes of *selector* are reserved (must be zero).

The *new attributes* operand contains the new values for the computational attribute bytes selected by *selector*. Refer to Retrieve Computational Attributes (RETCA) for more information on the structure of *new attributes*. If any of the reserved fields are not binary 0, a *scalar value invalid* (hex 3203) exception is signaled.

Note: Any floating-point operations currently on the value stack will be computed prior to changing the computational attributes.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

32 Scalar Specification

3203 Scalar Value Invalid

Set Data Pointer (SETDP)

Op Code (Hex)	Operand 1	Operand 2
0096	Receiver	Source

Operand 1: Data pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, or character variable array.

Bound program access
Built-in number for SETDP is 388. SETDP (space_addressability : address data_type_attributes : aggregate(7) OR See SETDPAT for format literal(7)) : data pointer

Description:

For non-bound programs: A data pointer is created and returned in the storage area specified by operand 1 and has the attributes and space addressability of the object specified by operand 2. Addressability is set to the low-order (leftmost) byte of the object specified by operand 2.

If operand 2 is a substring compound operand, the length attribute is set equal to the length of the substring. If operand 2 is a subscript compound operand, the attributes and addressability of the single array element specified are assigned to the data pointer. If operand 2 is an array, the attributes and addressability of the first element of the array are assigned to the data pointer. A data pointer can only be set to describe an element of a data array, not a data array in its entirety.

For bound (including service) programs: A data pointer is created and returned with the space addressability provided by the first operand and the attributes specified by the second operand. The attributes given to the data pointer include scalar type and scalar length.

For all programs: When the addressability in the data pointer is modified, the instruction signals the *space addressing violation* (hex 0601) exception when one of the following conditions occurs:

•

- When the space address to be stored in the pointer would have a negative offset value.
- When the offset would address an area beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated.

If the exception is signaled by this instruction for one of these reasons, the pointer is not modified by the instruction.

Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent cause the *space addressing violation* (hex 0601) exception to be signaled.

A data pointer cannot be set to address teraspace. Otherwise, an *unsupported space use* (hex 0607) exception is signaled.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Set Data Pointer Addressability (SETDPADR)

Op Code (Hex)	Operand 1	Operand 2
0046	Receiver	Source

Operand 1: Data pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, or character variable array.

Bound program access

```
Built-in number for SETDPADR is 389.
SETDPADR (
          data_pointer           : data pointer
          space_addressability   : address
) :   data pointer /* receiver */
```

Description:

For non-bound programs: The space addressability of the object specified for operand 2 is assigned to the data pointer specified by operand 1. If operand 1 contains a resolved data pointer, the data pointer's scalar attribute component is not changed by the instruction. If operand 1 contains an initialized but unresolved data pointer, the data pointer is resolved in order to establish the scalar attribute component

of the pointer. If operand 1 contains other than a resolved data pointer, the instruction creates and returns a data pointer in operand 1 with the addressability of the object specified for operand 2, and the instruction establishes the attributes as a character(1) scalar.

For bound (including service) programs: The space addressability specified by operand 2 is combined with attribute information from operand 1 and returned in a data pointer. If operand 1 contains a resolved data pointer, the data pointer's scalar attribute component is not changed by the instruction. If operand 1 contains an initialized but unresolved data pointer, the data pointer is resolved in order to establish the scalar attribute component of the pointer. If operand 1 contains other than a resolved data pointer, the instruction creates and returns a data pointer with the addressability specified by operand 2 and the attributes of a character(1) scalar.

For all programs: When the addressability is set into a data pointer, the *space addressing violation* (hex 0601) exception is signaled by the instruction only when the space address to be stored in the pointer has a negative offset value or if the offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated. If the exception is signaled for this reason, the pointer is not modified by the instruction. Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent of the space cause the *space addressing violation* (hex 0601) exception to be signaled.

A data pointer cannot be set to address teraspace. Otherwise, an *unsupported space use* (hex 0607) exception is signaled.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Set Data Pointer Attributes (SETDPAT)

Op Code (Hex)	Operand 1	Operand 2
004A	Receiver	Attributes

Operand 1: Data pointer.

Operand 2: Character(7) scalar.

Description: The value of the character scalar specified by operand 2 is interpreted as an encoded representation of an attribute set that is assigned to the attribute portion of the data pointer specified by

operand 1. The addressability portion of the data pointer is not modified. If operand 1 contains an initialized but unresolved data pointer, the data pointer is resolved in order to establish the addressability in the pointer. The attributes specified by the instruction are then assigned to the data pointer. If operand 1 does not contain a data pointer at the initiation of the instruction's execution, a *pointer does not exist* (hex 2401) exception or *pointer type invalid* (hex 2402) exception is signaled.

The format of the attribute set is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Data pointer attributes	Char(7)	
0	0		Scalar type	Char(1)
			Hex 00 = Signed binary	
			Hex 01 = Floating-point	
			Hex 02 = Zoned decimal	
			Hex 03 = Packed decimal	
			Hex 04 = Character	
			Hex 06 = Onlyns	
			Hex 07 = Onlys	
			Hex 08 = Either	
			Hex 09 = Open	
			Hex 0A = Unsigned binary	
1	1		Scalar length	Bin(2)
			If binary:	
1	1		Length (only 2, 4 or 8 for binary)	Bits 0-
			If floating-point:	
1	1		Length (only 4 or 8 for floating-point)	Bits 0-
			If zoned decimal or packed decimal:	
1	1		Fractional digits (F)	Bits 0-
1	1		Total digits (T) (where 1 <= T <= 63, 0 <= F <= T)	Bits 8-
			If character:	
1	1		Length (L, where 1 <= L <= 32,767)	Bits 0-
			If Onlyns:	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
1	1		Length (L, where 1 <= L <= 16,383) L is the number of double-byte characters	Bits 0-15
			If Onlys:	
1	1		Length (L, where 2 <= L <= 32,766) • L is the number of bytes • L is even • L includes any SO and SI characters	Bits 0-15
			If Either:	
1	1		Length (L, where 1 <= L <= 32,766) • L is the number of bytes • L includes any SO and SI characters	Bits 0-15
			If Open:	
1	1		Length (L, where 1 <= L <= 32,766) • L is the number of bytes • L includes any SO and SI characters	Bits 0-15
3	3		Reserved (binary 0)	Bin(4)
7	7	— End —		

Support for usage of a data pointer describing an Onlyns, Onlys, Either, or Open scalar value is limited to the Copy Extended Characters Left Adjusted With Pad instruction (CPYECLAP). Usage of such a data pointer defined value on any other instruction is not supported and results in the signaling of the *scalar type invalid* (hex 3201) exception.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Set Instruction Pointer (SETIP)

Op Code (Hex)	Operand 1	Operand 2
1022	Receiver	Branch target

Operand 1: Instruction pointer.

Operand 2: Instruction number, relative instruction number, or branch point.

Description: The value of the *branch target* (operand 2) is used to set the value of the instruction pointer specified by operand 1. The instruction number indicated by the branch target must provide the address of an instruction within the program containing the Set Instruction Pointer instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Set Invocation Exit (SETIEXIT)

Op Code (Hex)	Operand 1	Operand 2
0252	Invocation exit program	Argument list

Operand 1: System pointer.

Operand 2: Operand list or null.

Description: This instruction allows the external entry point of the program specified by operand 1 to be given control when the requesting invocation is destroyed due to normal exception handling actions, or due to any thread termination. Normal exception handling actions are considered to be those actions performed by the Return From Exception (RTNEXCP) or the Signal Exception (SIGEXCP) instructions.

Operand 1 is a system pointer addressing the program that is to receive control. The operand 1 system pointer must be in either the static or automatic storage of the program invoking this instruction.

Operand 2 specifies an operand list that identifies the arguments to be passed to the invocation exit program being called. If operand 2 is null, no arguments are passed to the invocation.

No operand verification takes place when this instruction is executed. Nor are copies made of the operands, so changes made to the operand values after execution of this instruction will be used during later operand verification. Operand verification occurs on the original form of the operands when the invocation exit program is invoked. At that time execute authorization verification to the invocation exit program and any contexts referenced for materialization take place. Also, materialization lock enforcement occurs to contexts referenced for materialization.

Operand 1 must point to a non-bound program or a bound program. Operand 1 should not point to a bound service program or a Java(TM) program or else an error will occur when an attempt is made to invoke the invocation exit program.

If an invocation exit program currently exists for the requesting invocation, it is replaced, and no exception is signaled. The invocation exit set by this instruction is implicitly cleared when the invocation exit program is given control, or the program which set the invocation exit completes execution.

If any invocations are to be destroyed due to normal exception handling actions, then those invocation exit programs associated with the invocations to be destroyed are given control before execution proceeds to the signaled exception handler.

The invocation exit program that is being destroyed is terminated, and its associated invocation execution is terminated. Termination of invocations due to a previous Signal Exception instruction, a Return From Exception instruction, a process termination, or a thread termination, is then resumed.

If a process phase is terminated and the process was not in termination phase, then the invocations are terminated. Invocation exit programs set for the terminated invocations are allowed to run. If an invocation to be terminated is an invocation exit program, then the following occurs:

•
- If an invocation exit has been set for this invocation exit, it is allowed to run.
- The invocation exit is terminated and the associated invocation is terminated (the invocation exit is not reinvoked).

Invocation exit programs for the remaining invocations to be terminated are then allowed to run.

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2208 Object Compressed

220B Object Not Available

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Set Invocation Flags (SETINVF)

Bound program access
Built-in number for SETINVF is 5. SETINVF (set_mask : unsigned binary(4) value which specifies the invocation flags to be set)

Description: Operand 1 selects which invocation flags are to be set. Only the invocation flags that are ″writeable″ can be set. Any ″read-only″ flags selected by the stack value are unchanged. The operation is performed by doing a bit-wise Boolean **or** of the 16 writeable status bits with the low-order two bytes of the *set mask* operand, and then replacing the writeable status bits with the result of this **or**.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

Set Object Pointer from Pointer (SETOBPFP)

Bound program access
Built-in number for SETOBPFP is 455. SETOBPFP (receiver : address of object pointer source_pointer : address of system pointer)

Description: The instruction returns an object pointer to the XOM object addressed by the *source pointer*. Upon return, the object pointer will address the XOM object.

The *source pointer* must address a XOM object or a *pointer addressing invalid object type* (hex 2403) exception will be signalled.

The *source pointer* must be a resolved system pointer or a *pointer not resolved* (hex 2404) exception will be signalled.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2403 Pointer Addressing Invalid Object Type

2404 Pointer Not Resolved

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Set Space Pointer (SETSPP)

Op Code (Hex)	Operand 1	Operand 2
0082	Receiver	Source

Operand 1: Space pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, or pointer data object.

Description: A space pointer is returned in operand 1 and is set to address the lowest order (leftmost) byte of the byte string identified by operand 2.

When the addressability is set in a space pointer, the instruction signals the *space addressing violation* (hex 0601) exception when the offset addresses beyond the largest space allocatable in the object or when the space address to be stored in the pointer has a nonpositive offset value. This offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated. If the exception is signaled for this reason, the pointer is not modified by the instruction. Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent of the space cause the *space addressing violation* (hex 0601) exception to be signaled.

If a pointer data object specified for operand 2 contains a data pointer value upon execution of the instruction, the addressability is set to the pointer storage form rather than to the scalar described by the data pointer value. The variable scalar references allowed on operand 2 cannot be described through a data pointer value.

The *object destroyed* (hex 2202) exception, the *parameter reference violation* (hex 0801) exception, and the *pointer does not exist* (hex 2401) exception are not signaled when operand 1 is a space pointer machine object and operand 2 is based on a space pointer machine object. This occurs when the basing space pointer machine object for operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition which existed for operand 2. The appropriate exception condition is signaled for either pointer upon a subsequent attempt to reference the space data the pointer addresses.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Set Space Pointer from Pointer (SETSPPFP)

Op Code (Hex)	Operand 1	Operand 2
0022	Receiver	Source pointer

Operand 1: Space pointer.

Operand 2: Data pointer, system pointer, or space pointer.

Bound program access

```
Built-in number for SETSPPFP is 141.
SETSPPFP (
          source_pointer   : system pointer OR
                             data pointer OR
                             space pointer(16)
) :   address /* receiver */
```

The *source pointer* operand corresponds to operand 2 on the SETSPPFP operation. The return value corresponds to operand 1 after the function completes.

Description: A space pointer is returned in operand 1 with the addressability to a space from the pointer specified by operand 2.

The meaning of the pointers allowed for operand 2 is as follows:

Pointer	Meaning
Data pointer or space pointer	The space pointer returned in operand 1 is set to address the same byte addressed by the operand 2 source pointer.
System pointer	The space pointer returned in operand 1 is set to address the first byte of the space associated with the system object addressed by the system pointer for operand 2. The space addressed is either the created system space or an associated space for the system object addressed by the system pointer. If the operand 2 system pointer addresses a system object with no associated space, the *invalid space reference* (hex 0605) exception is signaled.

The *object destroyed* (hex 2202) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 and operand 2 are space pointer machine objects. This occurs when operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition that existed for operand 2. The appropriate exception condition will be signaled for either pointer when a subsequent attempt is made to reference the space data that the pointer addresses.

If operand 2 is a system pointer to a XOM object, the *object not eligible for operation* (hex 2204) exception will be signalled.

Warning: Temporary Level 3 Header

Authorization Required

•

• Space authority
  –
  – Operand 2 (if a system pointer)
• Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

0605 Invalid Space Reference

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

16 Exception Management

1604 Retry/Resume Invalid

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2204 Object Not Eligible for Operation

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Set Space Pointer Offset (SETSPPO)

Op Code (Hex)	Operand 1	Operand 2
0092	Receiver	Source

Operand 1: Space pointer.

Operand 2: Binary scalar.

Description: The value of the binary scalar specified by operand 2 is assigned to the offset portion of the space pointer identified by operand 1. The space pointer continues to address the same space object.

Operand 1 must contain a space pointer; otherwise, a *pointer type invalid* (hex 2402) exception is signaled.

When operand 1 points into teraspace, the pointer offset value might not be set correctly because there is no binary field large enough to hold a full teraspace offset. However, an exception may or may not be signaled. Note that STSPPO signals an exception when attempted on a space pointer containing a teraspace address, so there can be no expectation that a stored offset can later be used to set a teraspace offset. However, an alternative to SETSPPO for handling some teraspace values is to use a pointer to the start of the teraspace allocation and then add in an offset computed using the SUBSPPFO instruction. Offsets within one teraspace allocation can be used to set a space pointer in this way when the specific allocation size is known to be smaller than the maximum value of a binary variable. See STSPPO for more information.

When the addressability in the space pointer is modified, the instruction signals a *space addressing violation* (hex 0601) exception when one of the following conditions occurs:

-
- The space address to be stored in the pointer has a negative offset value.
- The offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated.

If the exception is signaled by this instruction for this reason, the pointer is not modified by the instruction.

Attempts to use a pointer whose offset value lies: between the currently allocated extent of the space and the maximum allocatable extent, or whose offset is outside all teraspace allocations, cause the *space addressing violation* (hex 0601) exception to be signaled.

Warning: Temporary Level 3 Header

Authorization Required

-
- None

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Set Space Pointer with Displacement (SETSPPD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0093	Receiver	Source	Displacement

Operand 1: Space pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, or pointer data object.

Operand 3: Binary scalar.

Description: A space pointer is returned in operand 1 and is set to the space addressability of the lowest (leftmost) byte of the object specified for operand 2 as modified algebraically by an integer displacement specified by operand 3. Operand 3 can have a positive or negative value. I.e.

```
Operand 1 = Address_of(Operand 2) + Operand 3
```

When the addressability is set in a space pointer, the instruction signals the *space addressing violation* (hex 0601) exception when the space address to be stored in the pointer has a negative offset value or when the offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated. If the exception is signaled for this reason, the pointer is not modified by the instruction. Attempts to use a pointer whose offset value lies: between the currently allocated extent of the space and the maximum allocatable extent of the space, or whose offset is outside all teraspace allocations, cause the *space addressing violation* (hex 0601) exception to be signaled.

If a pointer data object specified for operand 2 contains a data pointer value upon execution of the instruction, the addressability is set to the pointer storage form rather than to the scalar described by the data pointer value. The variable scalar references allowed on operand 2 cannot be described through a data pointer value.

The *object destroyed* (hex 2202) exception, the *parameter reference violation* (hex 0801) exception, and the *pointer does not exist* (hex 2401) exception are not signaled when operand 1 is a space pointer machine object and operand 2 is based on a space pointer machine object. This occurs when the basing space pointer machine object for operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition which existed for operand 2. The appropriate exception condition is signaled for either pointer upon a subsequent attempt to reference the space data the pointer addresses.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2201 Object Not Found
- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

36 Space Management

- 3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Set System Pointer from Pointer (SETSPFP)

Op Code (Hex)	Operand 1	Operand 2
0032	Receiver	Source pointer

Operand 1: System pointer.

Operand 2: System pointer, space pointer, data pointer, instruction pointer, label pointer or object pointer.

Bound program access

```
Built-in number for SETSPFP is 142.
SETSPFP (
         source_pointer   : system pointer OR
                            data pointer OR
                            space pointer(16) OR
                            label pointer OR
                            object pointer
) :   system pointer /* receiver */
```

The *source pointer* operand corresponds to operand 2 on the SETSPFP operation. The return value corresponds to operand 1 after the function completes.

Description: This instruction returns a system pointer to the system object addressed by the supplied pointer.

If operand 2 is a system pointer, then a system pointer addressing the same object is returned in operand 1 containing the same authority as the input pointer.

If operand 2 is a space pointer or a data pointer, then a system pointer addressing the system object associated with the space addressed by operand 2 is returned in operand 1. The system object associated with machine supplied spaces used for automatic, static, activation group-based heap space and handle-based heap space storage is the process control space (PCS) object with which they are affiliated. The system object associated with a teraspace is the PCS object used by the process which contains the currently executing thread.

If operand 2 is an instruction pointer or label pointer, then a system pointer addressing the program system object that contains the instruction or label addressed by operand 2 is returned in operand 1.

If operand 2 is an unresolved system pointer or data pointer, the pointer is resolved first.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

  –

  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialization
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0604 External Data Object Not Found

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Signal Exception (SIGEXCP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3-4
SIGEXCP 10CA		Attribute template	Exception data	
SIGEXCPB 1CCA	Branch options	Attribute template	Exception data	Branch targets
SIGEXCPI 18CA	Indicator options	Attribute template	Exception data	Indicator targets

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3-4:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction signals a new exception or resignals an existing exception to the thread. Optionally, the instruction branches to one of the specified targets based on the results of the signal and

the selected branch options in the extender field, or it sets indicators based on the results of the signal. The signal is presented starting at the invocation identified in the signal template.

The template identified by operand 1 specifies the signal option and starting point. It must be 16-byte aligned in the space with the following format.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Target invocation address	Space pointer or Invocation pointer	
16	10	Signal option	Char(1)	
16	10		Signal/resignal option	Bit 0
			0 = Signal new exception.	
			1 = Resignal currently handled exception (valid only for an external exception handler).	
16	10		Invoke PDEH (process default exception handler) option	Bit 1
			0 = Invoke PDEH if no exception description found for invocation.	
			1 = Do not invoke PDEH if no exception description found for invocation (ignore if base invocation entry specified).	
16	10		Exception description search control	Bit 2
			0 = Exception description search control not present	
			1 = Exception description present	
16	10		Reserved (binary 0)	Bits 3-7
17	11	Reserved (binary 0)	Char(1)	
18	12	First exception description to search	Bin(2)	
20	14	—- End —-		

The **target invocation address** pointer uniquely identifies the invocation to which the exception is to be signalled. Signalling directly to the PDEH can not be accomplished via this instruction. If the *target invocation address* pointer locates neither a valid invocation entry nor the base invocation entry, the *invalid invocation address* (hex 1603) exception is signaled.

The invocation which issued this instruction will be checked to ensure it has the proper authority to send an exception message to the target invocation. If the authority check fails, *activation group access violation* (hex 2C12) exception will be signaled. If the program associated with the invocation has defined an exception description to handle the exception, the specified action is taken; otherwise, the PDEH is invoked unless the **invoke PDEH option** bit is 1 (the exception is considered ignored). If the base invocation entry is addressed instead of an existing invocation, the PDEH will be invoked.

A change has been made to the way in which exception handlers are determined for bound programs. The following description relates only to the invocation of exception handlers related to non-bound programs. In both instances the actions of signalling and handling have been broken apart.

Note:

Exception descriptions of an invocation are searched in ascending ODT number sequence. If the **exception description search control** specified *exception description search control not present*, the search begins with the first exception description defined in the ODT. Otherwise, the **first exception description to search** value identifies the relative number of the exception description to be used to start the search. The value must be a nonzero positive binary number determined by the order of definition of exception descriptions in the ODT. This value is also returned by the Sense Exception Description (SNSEXCPD) instruction. A value of 1 indicates that the first exception description in the invocation is to be used to begin the search. If the value is greater than the number of exception descriptions for the invocation, the *template value invalid* (hex 3801) exception is signaled.

If an **exception ID** in an exception description corresponds to the signaled exception, the corresponding **compare values** are verified. If the *compare value length* in the exception description is less than the *compare value length* in the signal template, the length of the compare value in the exception description is used for the match. If the *compare value length* in the exception description is greater than the *compare value length* in the signal template, an automatic mismatch results. Machine-signaled exceptions have a 4-byte compare value of binary 0's.

An exception description may monitor for an exception with a generic ID as follows:

Hex 0000 =

Any signaled exception ID results in a match.

Hex nn00 =

Any signaled exception ID in class nn results in a match.

Hex nnmm =

The signaled exception ID must be exactly nnmm in order for a match to occur.

An exception description may be in one of five states, each of which determines an action to be taken when the match criteria on the exception ID and compare value are met.

IGNORE

No exception handling occurs. The Signal Exception instruction is assigned a resultant condition of ignored. If a corresponding branch or indicator setting is present, that action takes place.

DISABLE

The exception description is bypassed, and the search for a monitor continues with the next exception description defined for the invocation.

RESIGNAL

The search for a monitoring exception description is to be reinitiated at the preceding invocation. A resignal from the initial invocation in the thread results in the

When this instruction is invoked with the *resignal* option, all invocations up to, but not including, the interrupted invocation are cancelled and the message is signalled to the next oldest invocation in the stack. This implies that the Return from Exception (RTNEXCP) instruction can no longer return to the invocation that issued the resignal request. Any cancel handlers set for the cancelled invocations will be given control before execution proceeds in the signaled exception handler.

If a failure to invoke an external exception handler or an invocation exit occurs, a failure to invoke program event is signaled. For each destroyed invocation, the invocation count in the corresponding activation entry (if any) is decremented by 1.

The template identified by operand 2 must be 16-byte aligned in the space. It specifies the exception-related data to be passed with the exception signal. The format of the *exception data* is the same as that returned by the Retrieve Exception Data (RETEXCPD) instruction. The format is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Template size	Char(8)	
0	0		Number of bytes of data to be signaled (must be at least 48 bytes)	Bin(4)
4	4		Number of bytes available for materialization	Bin(4) +
8	8	Exception identification	Char(2)	
10	A	Compare value length (maximum of 32 bytes)	Bin(2)	
12	C	Compare value	Char(32)	
44	2C	Reserved	Char(4) +	
48	30	Exception specific data	Char(*)	
*	*	— End —		

Note: Fields shown here with a plus sign (+) are ignored by the instruction.

Operand 2 is ignored if *signal/resignal option* is *resignal* because the exception-related data is the same as for the exception currently being processed; however, it must be specified when signaling a new exception.

The maximum size for *exception specific* data that is to accompany an exception signaled by the Signal Exception instruction is 65,503 bytes, including the standard exception data.

The following parameters will be given the following default values:

Message status - log message + retain + action pending

Initial monitor priority - 64

Interrupt class mask - Message generated by Signal Exception instruction

Source invocation - invocation issuing SIGEXCP instruction

Resultant Conditions:

•

- Exception ignored.
- Exception deferred.

Warning: Temporary Level 3 Header

Authorization Required

The invocation which originated the exception must have proper activation group access to the target invocation. The following algorithm is used to determine this access.

1. The invocation which invoked the SIGEXCP instruction must have access to the invocation identified as the Originating Invocation.
2. The Originating Invocation must have access to the invocation identified as the Source Invocation or to the invocation directly called by the Source invocation.
3. The Originating Invocation must have access to the invocation identified as the Target Invocation or to the invocation directly called by the Target Invocation.

If any of the access checks fail then an *activation group access violation* (hex 2C12) exception will be signaled.

Lock Enforcement

-
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

16 Exception Management

1602 Exception State of Thread Invalid

1603 Invalid Invocation Address

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C12 Activation Group Access Violation

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3802 Template Size Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Sine (SIN)

Bound program access
Built-in number for SIN is 398. SIN (source : floating point(8) value) : floating point(8) value which is the sine of the source value

Description: The sine of the numeric value of the *source* operand, whose value is considered to be in radians, is computed and the result is returned.

The result is in the range:

```
-1 <= SIN(source) <= 1
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Sine Hyperbolic (SINH)

Bound program access
Built-in number for SINH is 407. SINH (source : floating point(8) value) : floating point(8) value which is the sine hyperbolic of the source value

Description: The sine hyperbolic of the numeric value of the *source* operand is computed and the result (in radians) is returned.

The result is in the range:

```
-infinity <= SINH(source) <= +infinity
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Store and Set Computational Attributes (SSCA)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
107B	Receiver	Source	Controls

Operand 1: Character(5) variable scalar.

Operand 2: Character(5) scalar or null.

Operand 3: Character(5) scalar or null.

Description: This instruction stores and optionally sets the attributes for controlling computational operations for the thread this instruction is executed in.

The *receiver* is assigned the values that each of the computational attributes had at the start of execution of the instruction. It has the same format and bit assignment as the *source*.

The *source* specifies new values for the computational attributes for the thread. The particular computational attributes that are selected for modification are determined by the *controls* operand. The *source* operand has the following format:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Floating-point exception masks	Char(2)
		0 = Disabled (exception is masked)	
		1 = Enabled (exception is unmasked)	

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0		Reserved (binary 0)	Bits 0-9
0	0		Floating-point overflow	Bit 10
0	0		Floating-point underflow	Bit 11
0	0		Floating-point zero divide	Bit 12
0	0		Floating-point inexact result	Bit 13
0	0		Floating-point invalid operand	Bit 14
0	0		Reserved (binary 0)	Bit 15
2	2	Floating-point exception occurrence flags	Char(2)	
		0 = Exception has not occurred		
		1 = Exception has occurred		
2	2		Reserved (binary 0)	Bits 0-9
2	2		Floating-point overflow	Bit 10
2	2		Floating-point underflow	Bit 11
2	2		Floating-point zero divide	Bit 12
2	2		Floating-point inexact result	Bit 13
2	2		Floating-point invalid operand	Bit 14
2	2		Reserved (binary 0)	Bit 15
4	4	Modes	Char(1)	
4	4		Reserved	Bit 0
4	4		Floating-point rounding mode	Bits 1-2
			00= Round toward positive infinity	
			01= Round toward negative infinity	
			10= Round toward zero	
			11= Round to nearest (default)	
4	4		Reserved	Bits 3-7
5	5	—- End —-		

If any of the reserved fields are not binary 0, a *scalar value invalid* (hex 3203) exception is signaled.

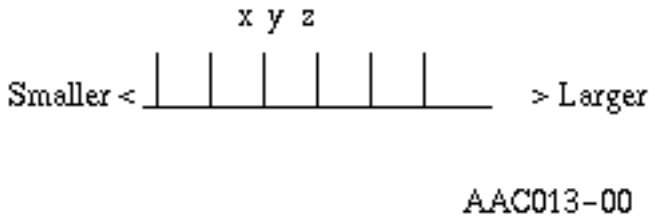
The *controls* operand is used to select those attributes that are to be set from the bit values of the *source* operand. The format of the *controls* is the same as that for the *source*. A value of one for a bit in *controls* indicates that the corresponding computational attribute for the thread is to be set from the value of that bit of the *source*. A value of zero for a bit in *controls* indicates that the corresponding computational attribute for the thread is not to be changed, and will retain the value it had prior to this instruction. For an attribute controlled by a multiple-bit field, such as the rounding modes, all of the bits in the field must be ones or all must be zeros. A mixture of ones and zeros in such a field results in a *scalar value invalid* (hex 3203) exception.

If the *source* and *controls* operands are both null, the instruction will just return the current computational attributes. If the *source* is specified, the computational attributes of the thread are modified under control of the *controls* operand. If the *source* operand is specified and the *controls* operand is null, the instruction will change all of the computational attributes to the values specified in the *source*.

With the **floating-point exception masks** field, it is possible to unmask/mask the exception processing and handling for each of the five floating-point exceptions that are maskable. If an exception that is unmasked occurs, then the corresponding *floating point exception occurrence* bit is set, and the exception is signaled. If an exception that is masked occurs, the exception is not signaled, but the *floating point exception occurrence* flag is still set to indicate the occurrence of the exception.

The **floating-point exception occurrence** flag for each exception may be set or cleared by this instruction from the *source* operand under control of the *controls* operand.

Unless specified otherwise by a particular instruction, or precluded due to implicit conversions, all floating-point operations are performed as if correct to infinite precision, and then rounded to fit in a destinations format while potentially signaling an exception that the result is inexact. To allow control of the floating-point rounding operations performed within a thread, four **floating-point rounding modes** are supported. Assume y is the infinitely precise number that is to be rounded, bracketed most closely by x and z, where x is the largest representable value less than y and z is the smallest representable value greater than y. Note that x or z may be infinity. The following diagram shows this relationship of x, y, and z on a scale of numerically progressing values where the vertical bars denote values representable in a floating-point format.



Given the above, if y is not exactly representable in the receiving field format, the rounding modes change y as follows:

Round to nearest with round to even in case of a tie is the default rounding mode in effect upon the initiation of a thread. For this rounding mode, y is rounded to the closer of x or z. If they are equally close, the even one (the one whose least significant bit is a zero) is chosen. For the purposes of this mode of rounding, infinity is treated as if it was even. Except for the case of y being rounded to a value of infinity, the rounded result will differ from the infinitely precise result by at most half of the least significant digit position of the chosen value. This rounding mode differs slightly from the decimal round algorithm performed for the optional round form of an instruction. This rounding mode would round a value of 0.5 to 0, where the decimal round algorithm would round that value to 1.

Round toward positive infinity indicates directed rounding upward is to occur. For this mode, y is rounded to z.

Round toward negative infinity indicates directed rounding downward is to occur. For this mode, y is rounded to x.

Round toward zero indicates truncation is to occur. For this mode, y is rounded to the smaller (in magnitude) of x or z.

Arithmetic operations upon infinity are exact. Negative infinity is less than every finite value, which is less than positive infinity.

The computational attributes are set with a default value upon thread initiation. The default attributes are as follows:

-
- The *floating-point inexact result* exception is masked. The other floating-point exceptions are unmasked.
- All *floating point occurrence bits* have a zero value.
- *Round to the nearest* rounding mode.

These attributes can be modified by a program executing this instruction. The new attributes are then in effect for the program executing this instruction and for programs invoked subsequent to it unless changed through another execution of this instruction. External exception handlers and invocation exit routines are invoked with the same attributes as were last in effect for the program invocation they are related to. Event handlers do not really relate to another invocation in the thread. As such, they are invoked with the attributes that were in effect at the point the thread was interrupted to handle the event.

Upon return to the invocation of a program from subsequent program invocations, the computational attributes, other than *floating point exception occurrence* attributes, are restored to those that were in effect when the program gave up control. The *floating point exception occurrence* attributes are left intact reflecting the occurrence of any floating-point exceptions during the execution of subsequent invocations.

Internal exception handlers execute under the invocation of the program containing them. As such, the above discussion of how computational attributes are restored upon returning from an external exception handler does not apply. The execution of an internal exception handler occurs in a manner similar to the execution of an internal subroutine invoked through the Call Internal (CALLI) instruction. If the internal exception handler modifies the attributes, the modification remains in effect for that invocation when the exception handler completes the exception.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Store Parameter List Length (STPLLEN)

Op Code (Hex)	Operand 1
0241	Length

Operand 1: Binary variable scalar.

Description: A value is returned in operand 1 that represents the number of parameters associated with the invocation's external entry point for which arguments have been passed on the preceding Call External (CALLX) or Transfer Control (XCTL) instruction.

The value can range from 0 (no parameters were received) to the maximum size possible for the parameter list associated with the external entry point.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Store Space Pointer Offset (STSPPO)

Op Code (Hex)	Operand 1	Operand 2
00A2	Receiver	Source

Operand 1: Binary variable scalar.

Operand 2: Space pointer.

Description: The offset value of the space pointer referenced by operand 2 is stored in the binary variable scalar defined by operand 1.

If operand 2 does not contain a space pointer, a *pointer does not exist* (hex 2401) exception is signaled.

If operand 2 points to teraspace, an *unsupported space use* (hex 0607) exception is signaled. This is necessary because no binary variable is large enough to contain an arbitrary teraspace offset. However, to retrieve a teraspace offset value within some specific allocation that is known to be smaller than the maximum value of a binary result variable, the SUBSPPFO instruction can be used. A pointer to the start of the teraspace allocation can be subtracted from a pointer to the current location within the allocation that is being referenced, for example.

If binary *size* (hex 0C0A) exceptions are to be signaled either because the program creation attribute indicated to do so or because a translator directive indicated to do so, they will be signalled under the following conditions:

-
- The offset value is greater than 32,767 and operand 1 is a signed binary (2) scalar.
- The offset value is greater than 65,535 and operand 1 is an unsigned binary (2) scalar.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C0A Size

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Subtract Logical Character (SUBLC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
SUBLC 1027		Difference	Minuend	Subtrahend	
SUBLCI 1827	Indicator options	Difference	Minuend	Subtrahend	Indicator targets
SUBLCB 1C27	Branch options	Difference	Minuend	Subtrahend	Branch targets

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4-6:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-5]
SUBLCS 1127		Difference/Minuend	Subtrahend	
SUBLCIS 1927	Indicator options	Difference/Minuend	Subtrahend	Indicator targets
SUBLCBS 1D27	Branch options	Difference/Minuend	Subtrahend	Branch targets

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3-5:

•
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The unsigned binary value of the *subtrahend* operand is subtracted from the unsigned binary value of the *minuend* operand, and the result is placed in the *difference* operand.

If the short form is not used and if neither source operand is an immediate value, then operands 2 and 3 must be the same length. The length can be a maximum of 256 bytes. In the case that the short form is not used and operand 2 or 3 is an immediate operand, it is treated as a character value and extended on the right with hex 00 bytes to match the length of the other operand. The subtraction operation is performed as though the ones complement of the second operand and a low-order 1-bit were added to the first operand.

The result value is then placed (left-adjusted) into the receiver operand with truncating or padding taking place on the right. The pad value used in this instruction is a byte value of hex 00.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Resultant Conditions: The logical difference of the character scalar operands is:

•
- Zero with carry out of the high-order bit position
- Not-zero with carry
- Not-zero with no carry.

Authorization Required

•
- None

Lock Enforcement

•
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

20 Machine Support

2002 Machine Check

2003 Function Check

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Subtract Numeric (SUBN)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
SUBN 1047		Difference	Minuend	Subtrahend	
SUBNR 1247		Difference	Minuend	Subtrahend	
SUBNB 1C47	Branch options	Difference	Minuend	Subtrahend	Branch targets
SUBNBR 1E47	Branch options	Difference	Minuend	Subtrahend	Branch targets
SUBNI 1847	Indicator options	Difference	Minuend	Subtrahend	Indicator targets
SUBNIR 1A47	Indicator options	Difference	Minuend	Subtrahend	Indicator targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4-7:

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Warning: Temporary Level 3 Header

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
SUBNS 1147		Difference/Minuend	Subtrahend	
SUBNSR 1347		Difference/Minuend	Subtrahend	
SUBNBS 1D47	Branch options	Difference/Minuend	Subtrahend	Branch targets
SUBNBSR 1F47	Branch options	Difference/Minuend	Subtrahend	Branch targets
SUBNIS 1947	Indicator options	Difference/Minuend	Subtrahend	Indicator targets
SUBNISR 1B47	Indicator options	Difference/Minuend	Subtrahend	Indicator targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

•
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Caution: If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used or whenever the *assume coincident operand overlap* attribute has been specified in the program template. If the *assume coincident operand overlap* attribute has not been specified in the program template and indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Description: The *difference* is the result of subtracting the *subtrahend* from the *minuend.*

Operands can have floating-point, packed or zoned decimal, signed or unsigned binary type.

Source operands are the *minuend* and *subtrahend.* The receiver operand is the *difference.*

If operands have different types, source operands, *minuend* and *subtrahend,* are converted according to the following rules:

1. If any one of the operands has floating point type, source operands are converted to floating point type.
2. Otherwise, if any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.
3. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Minuend and *subtrahend* are subtracted according to their type. Floating point operands are subtracted using floating point subtraction. Packed decimal operands are subtracted using packed decimal subtraction. Unsigned binary subtraction is used with unsigned binary operands. Signed binary operands are subtracted using two's complement binary subtraction.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary subtractions execute faster than either packed decimal or floating point subtractions.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

For a decimal operation, alignment of the assumed decimal point takes place by padding with 0's on the right end of the source operand with lesser precision.

Floating-point subtraction uses exponent comparison and significand subtraction. Alignment of the binary point is performed, if necessary, by shifting the significand of the value with the smaller exponent to the right. The exponent is increased by one for each binary digit shifted until the two exponents agree.

The operation uses the length and the precision of the source and receiver operands to calculate accurate results. Operations performed in decimal are limited to 31 decimal digits in the intermediate result.

The subtract operation is performed according to the rules of algebra.

The result of the operation is copied into the *difference* operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the difference operand, aligned at the assumed decimal point of the *difference* operand, or both before being copied to it. For fixed-point operation, if significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For the optional round form of the instruction, specification of a floating-point receiver operand is invalid.

For floating-point operations involving a fixed-point receiver field, if nonzero digits would be truncated off the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point *difference* operand, if the exponent of the resultant value is either too large or too small to be represented in the difference field, the *floating-point overflow* (hex 0C06) exception or the *floating-point underflow* (hex 0C07) exception is signaled.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Size exceptions can be inhibited.

Resultant Conditions:

•

- Positive-The algebraic value of the numeric scalar *difference* is positive.
- Negative-The algebraic value of the numeric scalar *difference* is negative.
- Zero-The algebraic value of the numeric scalar *difference* is zero.
- Unordered-The value assigned a floating-point *difference* operand is NaN.

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C02 Decimal Data

0C03 Decimal Point Alignment

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0A Size

0C0C Invalid Floating-Point Conversion

0C0D Floating-Point Inexact Result

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Subtract Space Pointer Offset (SUBSPP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0087	Receiver pointer	Source pointer	Decrement

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Binary scalar.

Description: The value of the binary scalar specified by operand 3 is subtracted from the space address contained in the space pointer specified by operand 2; the result is stored in the space pointer identified by operand 1. I.e.

```
Operand 1 = Operand 2 - Operand 3
```

Operand 3 can have a positive or negative value. The space object that the pointer is addressing is not changed by the instruction. If operand 2 does not contain a space pointer, a *pointer type invalid* (hex 2402) exception is signaled.

When the addressability in the space pointer is modified, the instruction signals a *space addressing violation* (hex 0601) exception when one of the following conditions occurs, for any space except teraspace:

-
- The space address to be stored in the pointer has a negative offset value.
- The offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated.

If the exception is signaled by this instruction for one of these reasons, the pointer is not modified by the instruction.

In contrast, when modifying the addressability of a space pointer to teraspace, if the address computed either overflows or underflows the offset, the result is wrapped back within teraspace and no exception is signalled. However, since the size of teraspace and thus the size of the offset portion of a teraspace address is implementation-dependent, the wrapped result may vary between machine implementations.

Attempts to use a pointer whose offset value lies: between the currently allocated extent of the space and the maximum allocatable extent of the space, or whose offset is outside all teraspace allocations, cause the *space addressing violation* (hex 0601) exception to be signaled.

The *object destroyed* (hex 2202) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 and operand 2 are space pointer machine objects. This occurs when operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition that existed for operand 2. The appropriate exception condition will be signaled for either pointer when a subsequent attempt is made to reference the space data that the pointer addresses.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Subtract Space Pointers For Offset (SUBSPPFO)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0033	Offset difference	Minuend pointer	Subtrahend pointer

Operand 1: Binary(4) variable scalar.

Operand 2: Space pointer.

Operand 3: Space pointer.

Description: The offset portion of the space address contained in the operand 3 space pointer is subtracted from the offset of the space address contained in the space pointer specified by operand 2; the result is stored in the 4 byte binary scalar identified by operand 1.

The offsets for operands 2 and 3 are strictly unsigned values, while the operand 1 result can have a positive or negative value.

No check is made to determine that the space pointers point to the same space. In addition, the existence of the pointers is not checked except for pointers used as a base for the operands. When the space pointers point to different spaces, or exactly one of the pointer operands is subject to the pointer does not exist condition, the resulting value is undefined, but no exception is signaled for those conditions. However, if both operand 2 and operand 3 are subject to the pointer does not exist condition, the result value is zero.

If either operand 2 or operand 3 contains a pointer which is not a space pointer, a *pointer type invalid* (hex 2402) exception is signaled.

A *size* (hex 0C0A) exception can be signalled when the program attribute to *signal size exceptions* is in effect, under the following conditions:

-
- the operand 1 field is unsigned binary and the resulting value of the subtraction is negative
- the offset value produced by the subtraction is larger than the result field can contain.

The *object destroyed* (hex 2202) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 2 and operand 3 are space pointer

machine objects. This occurs when operand 2 or operand 3 contains an internal machine value that indicates one of these error conditions exists. Even if the corresponding exception is not signaled, the operand 1 value is undefined in those cases.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

0C Computation

0C0A Size

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Synchronize Shared Storage Accesses (SYNCSTG)

Op Code (Hex)	Operand 1
03E5	Action

Operand 1: Character (4) constant or unsigned binary (4) constant.

Bound program access

```
Built-in number for SYNCSTG is 617.
SYNCSTG (
         action   : unsigned binary (4) literal
)
```

Description: Enforces an ordering on shared storage accesses performed by the issuing thread. In this discussion, shared storage accesses are reads and writes from/to storage shared among multiple threads. The threads may be associated with the same or different processes.

The *action* field specifies what type of shared storage accesses (reads, writes, or both) are to be ordered.

-
- 0 = both reads and writes to shared storage performed by this thread will be ordered.
- 1 = only reads from shared storage performed by this thread will be ordered.
- 2 = only writes to shared storage performed by this thread will be ordered.

The affected shared storage accesses will be ordered in the sense that accesses appearing in the logical flow of the source code before the SYNCSTG will be guaranteed to be completed from the standpoint of the issuing thread before those appearing in the logical flow after the SYNCSTG. For instance, if two reads from two shared locations are separated by a SYNCSTG(0), then the second access will read a value

no less current than the first access. If two writes to two shared locations are separated by a SYNCSTG(0), then the first write will be available before the second write.

To completely enforce shared storage access ordering between two or more threads, it is necessary that all threads dependent on the access ordering — both readers and writers of the shared data — use appropriate SYNCSTG operations or some other synchronization mechanism such as locks, mutexes, semaphores, or data queues (this list is not intended to be exhaustive).

This instruction is only guaranteed to affect the ordering of shared storage accesses, and is necessary only when the accesses are to different shared storage locations and the semantics of the program depend on the ordering of the shared accesses between two or more threads. In addition, this instruction is only necessary in cases where other synchronization mechanisms (locks, mutexes, etc) are not being used to serialize access to the shared storage locations.

This instruction may have an associated performance penalty, so it is recommended that SYNCSTG be used conservatively when possible.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

Tangent (TAN)

Bound program access

`Built-in number for TAN is 402.`
`TAN (`
`  source  : floating point(8) value`
`) :  floating point(8) value which is the tangent of the source value`

Description: The tangent of the numeric value of the *source* operand, whose value is considered to be in radians, is computed and the result is returned.

The result is in the range:

```
-infinity <= TAN(source) <= +infinity
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Tangent Hyperbolic (TANH)

Bound program access

`Built-in number for TANH is 409.`
`TANH (`
`  source  : floating point(8) value`
`) :  floating point(8) value which is the tangent hyperbolic of the`
`  source value`

Description: The tangent hyperbolic of the numeric value of the *source* operand is computed and the result (in radians) is returned.

The result is in the range:

```
-1 <= TANH(source) <= +1
```

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

Test and Replace Bytes (TESTRPL)

Bound program access

```
Built-in number for TESTRPL is 413.
TESTRPL (
         source          : address of an aggregate which is the source
                           for tested bytes and the receiver for
                           replaced bytes
         source_length   : is an unsigned binary(4) value which
                           specifies the length of the source aggregate
                           being tested and optionally modified
         position        : address of an aggregate which provides
                           relative position information within the
                           replacement operand for byte values which
                           match source values
         replacement     : address of an aggregate which provides
                           replacement byte values for source values
                           which matched a position operand value
         other_length    : unsigned binary(4) value which specifies the
                           lengths of the relative position and
                           replacement bytes aggregates
)
```

Description: Bytes in the *source* are tested for matching values in the *position* aggregate. If a match is found, the byte value from the same relative offset within the *replacement* aggregate, as the matching value is within the *position* aggregate, is used to change the *source* byte value.

The operation proceeds byte by byte from left to right until each byte in the *source* has been tested and optionally modified.

Each byte of the *source* is compared with the individual byte values in the *position* aggregate. If a byte of equal value does not exist in the *position* aggregate, the *source* byte value is left unchanged. If a byte of equal value is found in the *position* aggregate, the corresponding byte in the same relative location within the *replacement* aggregate is used to modify the original *source* value. If a byte value in the *position* aggregate is duplicated, the first occurrence (leftmost) is used.

If any of the other operands overlap with the *source* aggregate but do not share all of the same bytes, results of this operation are unpredictable.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Test and Replace Characters (TSTRPLC)

Op Code (Hex)	Operand 1	Operand 2
10A2	Receiver	Replacement

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Description: The character string value represented by operand 1 is tested byte by byte from left to right. Until a byte with a value in the range of hex F1 to hex F9 (inclusive) is found, each byte that has a value outside that range is assigned a byte value equal to the leftmost byte of operand 2. Thus any byte to the left of the leftmost nonzero zoned decimal data value is replaced with the leftmost byte value of operand 2. Both operands must be character strings. Only the first character of the replacement string is used in the operation.

The operation stops when the first nonzero zoned decimal digit is found or when all characters of the *receiver* operand have been replaced.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Test Authority (TESTAU)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
TESTAU 10F7		Available authority template receiver	System object or object template	Required authority template	
TESTAUB 1CF7	Branch options	Available authority template receiver	System object or object template	Required authority template	Branch targets
TESTAUI 18F7	Indicator options	Available authority template receiver	System object or object template	Required authority template	Indicator targets

Operand 1: Character(2) variable scalar or null.

Operand 2: System pointer or space pointer data object.

Operand 3: Character(2) scalar.

Operand 4-5:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for TESTAU is 63.
TESTAU (
        available_authority_template_receiver   : address OR
                                                  null operand
        system_object_or_object_template        : address of system pointer OR
                                                  address of space pointer(16)
        required_authority_template             : address
) :   signed binary(4) /* return_code */
```

The return code will be set as follows:

Return code	Meaning
1	Authorized.
0	Not Authorized.

This built-in function is used to provide support for the branch and indicator forms of the TESTAU instruction. The user must specify code to process the *return code* and perform the desired branching or indicator setting.

Description: This instruction verifies that the object authorities and/or ownership rights specified by operand 3 are currently available to the thread for the object specified by operand 2.

If operand 1 is not null, all of the authorities and/or ownership specified by operand 3 that are currently available to the thread are returned in operand 1.

If an *object template* is not specified (i.e. operand 2 is a system pointer), then authority verification is performed relative to the invocation executing this instruction. If an *object template* is specified (i.e. operand 2 is a space pointer), then authority verification is performed relative to the invocation specified in the template. Specifying an invocation causes the invocations subsequent to it to be bypassed in the authority verification process. This has the influence of excluding the program adopted user profiles for any of these excluded invocations from acting as a source of authority to the authority verification process.

The required authorities and/or ownership are specified by the *required authority template* of operand 3. This template includes a test option that indicates whether all of the specified authorities are required or whether any one or more of the specified authorities is sufficient. This option can be used, for example, to test for operational authority by coding a template value of hex 0F01 in operand 3. Using the *any* option does not affect what is returned in operand 1. If operand 1 is not null and the *any* option is specified, all of the authorities specified by operand 3 that are available to the process are returned in operand 1.

If the required authority is available, one of the following occurs:

-
- Branch form indicated
  -
  - Conditional transfer of control to the instruction indicated by the appropriate branch target operand.
- Indicator form specified
  -

– The leftmost byte of each of the indicator operands is assigned the following values:
  - Hex F1- If the result of the test matches the corresponding indicator option
  - Hex F0- If the result of the test does not match the corresponding indicator option

If no branch options are specified, instruction execution proceeds to the next instruction. If operand 1 is null and neither the branch or indicator form is used, an *invalid operand type* (hex 2A06) exception is signaled.

The format for the *available authority template* (operand 1) is as follows: (1 = authorized)

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Authorization template	Char(2)	
0	0		Object control	Bit 0
0	0		Object management	Bit 1
0	0		Authorized pointer	Bit 2
0	0		Space authority	Bit 3
0	0		Retrieve	Bit 4
0	0		Insert	Bit 5
0	0		Delete	Bit 6
0	0		Update	Bit 7
0	0		Ownership (1 = yes)	Bit 8
0	0		Excluded	Bit 9
0	0		Authority list management	Bit 10
0	0		Execute	Bit 11
0	0		Alter	Bit 12
0	0		Reference	Bit 13
0	0		Reserved (binary 0)	Bits 14-15
2	2	—- End —-		

If operand 2 is a system pointer, it identifies the object for which authority is to be tested. If operand 2 is a space pointer, it provides addressability to the *object template*. The format for the optional *object template* is as follows:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Object template	Char(32)	
0	0		Relative invocation	Bin(2)
2	2		Reserved (binary 0)	Char(14)
16	10		System object	System pointer
32	20	—- End —-		

The **relative invocation** field in the *object template* identifies an invocation relative to the current invocation at which the authority verification is to be performed. The value of the relative invocation field must be less than or equal to zero. A value of zero identifies the current invocation, -1 identifies the prior invocation, -2, the invocation prior to that, and so on. A value larger than the number of invocations currently on the invocation stack or a positive value results in the signaling of the *template value invalid* (hex 3801) exception. The program adopted and propagated user profiles for the identified invocation and older invocations will be included in the authority verification process. Program adopted user profiles for invocations newer than the identified invocation will not be included in the authority verification process. If the current invocation is specified, its program adopted user profile is included whether or not it is to be propagated.

The **system object** field specifies a system pointer which identifies the object for which authority is to be tested.

The format for the *required authority template* (operand 3) is as follows: (1 = authorized)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Authorization template	Char(2)	
0	0		Object control	Bit 0
0	0		Object management	Bit 1
0	0		Authorized pointer	Bit 2
0	0		Space authority	Bit 3
0	0		Retrieve	Bit 4
0	0		Insert	Bit 5
0	0		Delete	Bit 6
0	0		Update	Bit 7
0	0		Ownership (1 = yes)	Bit 8
0	0		Excluded	Bit 9
0	0		Authority list management	Bit 10
0	0		Execute	Bit 11
0	0		Alter	Bit 12
0	0		Reference	Bit 13
0	0		Reserved (binary 0)	Bit 14
0	0		Test option	Bit 15
			0 = All of the above authorities must be present.	
			1 = Any one or more of the above authorities must be present.	
2	2	—- End —-		

This instruction will tolerate a damaged object referenced by operand 2 when the reference is a resolved pointer. The instruction will not tolerate damaged contexts or programs when resolving pointers. Damaged user profiles encountered during the authority verification processing result in the signaling of the *authority verification terminated due to damaged object* (hex 1005) exception.

Resultant Conditions:

•

- Authorized - the required authority is available.
- Unauthorized - the required authority is not available.

Warning: Temporary Level 3 Header

Authorization Required

•

- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize

  –

  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Test Bit in String (TSTBTS)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4]
TSTBTSB 1C0E	Branch options	Source	Offset	Branch targets
TSTBTSI 180E	Indicator options	Source	Offset	Indicator targets

Operand 1: Character scalar or numeric scalar.

Operand 2: Binary scalar.

Operand 3:

•

• *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.

- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access
Built-in number for TSTBTS is 1. TSTBTS (source : address offset : unsigned binary(4)) : signed binary(4) /* return_code */ The return code will be set as follows: **Return code** **Meaning** **1** Bit set. **0** Bit clear. This built-in function is used to provide support for the branch and indicator forms of the TSTBTS instruction. The user must specify code to process the *return_code* and perform the desired branching or indicator setting. The *offset* parameter must be between 0 and 65,535.

Description: Tests the bit of the *source* operand as indicated by the *offset* operand to determine if the bit is set or not set.

Based on the test, the resulting condition is used with the extender field to either

-
- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The *source* operand can be character or numeric. The leftmost bytes of the *source* operand are used in the operation. The *source* operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus one.

The *offset* operand indicates which bit of the *source* operand is to be tested, with a offset of zero indicating the leftmost bit of the leftmost byte of the *source* operand.

If an *offset* value less than zero or beyond the length of the string is specified a *scalar value invalid* (hex 3203) exception is signaled.

Resultant Conditions:

-
- Zero-The selected bit of the bit string *source* operand is zero.
- One-The selected bit of the bit string *source* operand is one.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Test Bits Under Mask (TSTBUM)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4, 5]
TSTBUMB 1C2A	Branch options	Source	Mask	Branch targets
TSTBUMI 182A	Indicator options	Source	Mask	Indicator targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3 [4, 5]

•

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: Selected bits from the leftmost byte of the *source* operand are tested to determine their bit values.

Based on the test, the resulting condition is used with the extender field to:

•

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The *source* and the *mask* operands can be character or numeric. The leftmost byte of each of the operands is used in the operands. The operands are interpreted as bit strings. The testing is performed bit by bit with only those bits indicated by the *mask* operand being tested. A 1-bit in the *mask* operand specifies that the corresponding bit in the source value is to be tested. A 0-bit in the *mask* operand specifies that the corresponding bit in the *source* value is to be ignored.

Resultant Conditions: The selected bits of the bit string *source* operand are all zeros, all ones, or mixed ones and zeros. A *mask* operand of all zeros causes a zero resultant condition.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Test Exception (TESTEXCP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3-4
TESTEXCP 104A		Receiver	Exception description	
TESTEXCPB 1C4A	Branch options	Receiver	Exception description	Branch options
TESTEXCPI 184A	Indicator options	Receiver	Exception description	Indicator options

Operand 1: Space pointer.

Operand 2: Exception description.

Operand 3-4:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The instruction tests the signaled status of the *exception description* specified in operand 2, and optionally alters the control flow or sets the specified indicators based on the test. Exception data is returned at the location identified by operand 1. The branch or indicator setting occurs based on the conditions specified in the extender field depending on whether or not the specified exception description is signaled.

Operand 2 is an *exception description* whose signaled status is to be tested. An exception can be signaled only if the referenced exception description is in the deferred state.

Operand 1 addresses a space into which the exception data is placed if an exception identified by the exception description has been signaled.

The template identified by operand 1 must be 16-byte aligned in the space and is formatted as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Template size	Char(8)	
0	0		Number of bytes provided for materialization	Bin(4)
4	4		Number of bytes available for materialization (0 if exception description is not signaled)	Bin(4)
8	8	Exception identification	Char(2)	
10	A	Compare value length (maximum of 32 bytes)	Bin(2)	
12	C	Compare value	Char(32)	
44	2C	Message reference key	Char(4)	
48	30	Exception-specific data	Char(*)	
*	*	Source invocation address	Invocation pointer or Null pointer	
*	*	Target invocation address	Invocation pointer	
*	*	Signaling program instruction address	UBin(2)	
*	*	Signaled program instruction address	UBin(2)	
*	*	Machine-dependent data	Char(10)	
*	*	—- End —-		

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

If the *exception description* is not in the signaled state, the *number of bytes available* for the materialization entry is set to binary 0's, and no other bytes are modified.

The **message reference key** field holds the architected value that uniquely identifies the exception message in a process queue space.

The **source invocation address** field will contain a null pointer value if the source invocation no longer exists when this instruction is executed.

The area beyond the *exception-specific data* area is extended with binary 0's so that pointers to program invocations are properly aligned.

If no branch options are specified, instruction execution proceeds at the instruction following the Test Exception instruction.

If the *exception data retention option*, from the *exception description*, is set to 1 (do not save), no data is returned by this instruction.

Resultant Conditions:

•

- Exception signaled.
- Exception not signaled.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

16 Exception Management

1601 Exception Description Status Invalid

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Test Extended Authorities (TESTEAU)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
TESTEAU 10FB		Available authority template receiver	Required authority template	Relative invocation	
TESTEAUB 1CFB	Branch options	Available authority template receiver	Required authority template	Relative invocation	Branch targets
TESTEAUI 18FB	Indicator options	Available authority template receiver	Required authority template	Relative invocation	Indicator targets

Operand 1: Character(8) variable scalar or null.

Operand 2: Character(8) scalar.

Operand 3: Binary(2) variable scalar or constant or null.

Operand 4-5:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for TESTEAU is 64.
TESTEAU (
         available_authority_template_receiver   : address OR
                                                   null operand
         required_authority_template             : address
         relative_invocation                     : address of signed binary(2) OR
                                                   null operand
) :   signed binary(4) /* return_code */
```

The return code will be set as follows:

Return code	Meaning
1	Authorized.
0	Not Authorized.

This built-in function is used to provide support for the branch and indicator forms of the TESTEAU operation. The user must specify code to process the *return code* and perform the desired branching or indicator setting.

Description: This instruction verifies that the privileged instructions and special authorities specified by operand 2 are currently available to the thread.

If operand 1 is not null, all of the privileged instructions and special authorities specified by operand 2 that are currently available to the thread are returned in operand 1.

Note: The term *authority verification* refers to the testing of the required privileged instruction and special authorities.

If operand 3 is null, the authority verification is performed relative to the invocation executing this instruction. If an operand 3 is specified, the authority verification is performed relative to the invocation specified. Specifying an invocation causes the invocations subsequent to it to be bypassed in the authority verification process. This has the influence of excluding the program adopted user profiles for any of these excluded invocations from acting as a source of authority to the authority verification process.

The required privileged instructions and special authorities are specified by the required authority template of operand 2.

If the required authority is available, one of the following occurs:

-
- Branch form indicated
  -
  - Conditional transfer of control to the instruction indicated by the appropriate branch target operand.
- Indicator form specified
  -
  - The leftmost byte of each of the indicator operands is assigned the following values:
    - Hex F1 - If the result of the test matches the corresponding indicator option
    - Hex F0 - If the result of the test does not match the corresponding indicator option

If no branch options are specified, instruction execution proceeds to the next instruction. If operand 1 is null and neither the branch or indicator form is used, an invalid operand type exception is signaled.

The format for the *available authority template receiver* (operand 1) is as follows: (1 = authorized)

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0	Authority template	Char(8)		
0	0		Privileged instruction template	Char(4)	
0	0		Create Logical Unit Description		Bit 0
0	0		Create Network Description		Bit 1
0	0		Create Controller Description		Bit 2
0	0		Create user profile		Bit 3
0	0		Modify user profile		Bit 4
0	0		Diagnose		Bit 5
0	0		Terminate machine processing		Bit 6
0	0		Initiate process		Bit 7
0	0		Modify Resource Management Control		Bit 8
0	0		Create Mode Description		Bit 9
0	0		Create Class of Service Description		Bit 10
0	0		Reserved (binary 0)		Bits 11-31
4	4		Special authority template	Char(4)	
4	4		All object		Bit 0
4	4		Load (unrestricted)		Bit 1
4	4		Dump (unrestricted)		Bit 2
4	4		Suspend (unrestricted)		Bit 3
4	4		Load (restricted)		Bit 4
4	4		Dump (restricted)		Bit 5
4	4		Suspend (restricted)		Bit 6
4	4		Process control		Bit 7
4	4		Reserved (binary 0)		Bit 8
4	4		Service		Bit 9
4	4		Auditor authority		Bit 10
4	4		Spool control		Bit 11
4	4		I/O system configuration		Bit 12
4	4		Reserved (binary 0)		Bits 13-23
4	4		Modify machine attributes		Bits 24-31
4	4			Group 2	E
4	4			Group 3	E
4	4			Group 4	E
4	4			Group 5	E
4	4			Group 6	E
4	4			Group 7	E
4	4			Group 8	E
4	4			Group 9	E
8	8	—- End —-			

The format for the *required authority template* (operand 2) is as follows: (1 = authorized)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Required authority	Char(8)	
0	0		Privileged instruction template	Char(4)

Offset Dec	Offset Hex	Field Name	Data Type and Length		
0	0		Create Logical Unit Description		Bit 0
0	0		Create Network Description		Bit 1
0	0		Create Controller Description		Bit 2
0	0		Create user profile		Bit 3
0	0		Modify user profile		Bit 4
0	0		Diagnose		Bit 5
0	0		Terminate machine processing		Bit 6
0	0		Initiate process		Bit 7
0	0		Modify Resource Management Control		Bit 8
0	0		Create Mode Description		Bit 9
0	0		Create Class of Service Description		Bit 10
0	0		Reserved (binary 0)		Bits 11-3
4	4		Special authority template	Char(4)	
4	4		All object		Bit 0
4	4		Load (unrestricted)		Bit 1
4	4		Dump (unrestricted)		Bit 2
4	4		Suspend (unrestricted)		Bit 3
4	4		Load (restricted)		Bit 4
4	4		Dump (restricted)		Bit 5
4	4		Suspend (restricted)		Bit 6
4	4		Process control		Bit 7
4	4		Reserved (binary 0)		Bit 8
4	4		Service		Bit 9
4	4		Auditor authority		Bit 10
4	4		Spool control		Bit 11
4	4		I/O system configuration - DAC		Bit 12
4	4		Reserved (binary 0)		Bits 13-2
4	4		Modify machine attributes		Bits 24-3
4	4			Group 2	Bit 24
4	4			Group 3	Bit 25
4	4			Group 4	Bit 26
4	4			Group 5	Bit 27
4	4			Group 6	Bit 28
4	4			Group 7	Bit 29
4	4			Group 8	Bit 30
4	4			Group 9	Bit 31
8	8	—- End —-			

The *relative invocation* operand (operand 3) identifies an invocation relative to the current invocation at which the authority verification is to be performed. The value of the *relative invocation* operand must be less than or equal to zero. A value of zero identifies the current invocation, -1 identifies the prior invocation, -2, the invocation prior to that, and so on. A value larger than the number of invocations currently on the invocation stack or a positive value results in the signaling of the *scalar value invalid* (hex 3203) exception.

An immediate value is not allowed for operand 3.

The program adopted and propagated user profiles for the identified invocation and older invocations will be included in the authority verification process. Program adopted user profiles for invocations newer than the identified invocation will not be included in the authority verification process. If the current invocation is specified, its program adopted user profile is included whether or not it is to be propagated.

Damaged user profiles encountered during the authority verification processing result in the signaling of the *authority verification terminated due to damaged object* (hex 1005) exception

Resultant Conditions:

•

- Authorized - the required authority is available.
- Unauthorized - the required authority is not available.

Warning: Temporary Level 3 Header

Authorization Required

•

- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize
  –
  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Test Initial Thread (TSTINLTH)

Op Code (Hex)	Operand 1
03A1	Result

Operand 1: Signed binary(4) variable scalar.

Bound program access
Built-in number for TSTINLTH is 509. TSTINLTH ( ) : signed binary(4) /* result */

Description: The *result* is set to indicate whether execution is within a process's initial thread or a secondary thread. A process's **initial thread** is the thread that is implicitly initiated by the machine when a process is initiated. Each process has a minimum of one thread associated with it. This is true regardless of whether the initiation phase, problem phase, or termination phase program of the initial thread is executing.

The returned *result* will have one of two possible values.

0	execution is within a secondary thread.
1	execution is within the initial thread.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Test Pending Interrupts (TESTINTR)

Bound program access
Built-in number for TESTINTR is 359. TESTINTR ( ) : An unsigned binary(4) with the set of pending thread interrupts is returned

Description: The set of pending interrupts is returned for the current thread. The format of the returned *pending interrupts* is defined as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Pending interrupts	UBin(4)	
0	0		Time slice end	Bit 0
0	0		Transfer lock	Bit 1
0	0		Asynchronous lock retry	Bit 2
0	0		Suspend process	Bit 3
0	0		Resume process	Bit 4
0	0		Modify resource management attributes	Bit 5
0	0		Modify process or thread attributes	Bit 6
0	0		Terminate machine processing	Bit 7
0	0		Terminate process or thread	Bit 8
0	0		Wait time-out	Bit 9
0	0		Event schedule	Bit 10
0	0		Thread operations between threads	Bit 11
0	0		Cancel long running MI instruction	Bit 12
0	0		Reserved (binary 0)	Bit 13
0	0		Deliver queue space message	Bit 14
0	0		Signal schedule	Bit 15
0	0		Reserved (binary 0)	Bits 16-31
4	4	—- End —-		

Note: Other alternatives for obtaining the pending interrupt information are option hex 20 or option hex 24 on the Materialize Process Attributes (MATPRATR) instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

36 Space Management

3601 Space Extension/Truncation

Test Performance Data Collection (TESTPDC)

Op Code (Hex)	Extender	Operand 1	Operand [2-3]
TESTPDCB 1C21	Branch options	Dummy scalar	Branch targets
TESTPDCI 1821	Indicator options	Dummy scalar	Indicator targets

Operand 1: Character(2) scalar or null.

Operand [2-3] :

•

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access
Built-in number for TESTPDC is 576. TESTPDC ( ) : signed binary(4) /* return_code */ The return code will be set as follows: **Return code** — **Meaning** **0** — The thread is not in an active Performance Data Collector (PDC) trace collection. **1** — The thread is in an active Performance Data Collector (PDC) trace collection. This built-in function is used to provide support for the branch and indicator forms of the TESTPDC instruction. The user must specify code to test the *return code* and perform the desired branching or indicator setting.

Description: A test is performed to determine whether or not the thread is in an active Performance Data Collector (PDC) trace collection.

One of the following occurs:

•

- Branch form indicated
  –
  – Conditional transfer of control to the instruction indicated by the appropriate branch target operand.
- Indicator form specified
  –
  – The leftmost byte of each of the indicator operands is assigned the following values:

Hex F1-	If the result of the test matches the corresponding indicator option
Hex F0-	If the result of the test does not match the corresponding indicator option

If no branch options are specified, instruction execution proceeds to the next instruction.

Resultant Conditions:

•

- True - The thread is in an active Performance Data Collector (PDC) trace collection.
- False - The thread is not in an active Performance Data Collector (PDC) trace collection.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

08 Argument/Parameter

0801 Parameter Reference Violation

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2C Program Execution

2C04 Branch Target Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Test Pointer (TESTPTR)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
01D3	Source pointer	Test type	Test result

Operand 1: Pointer data object.

Operand 2: Character(1) scalar immediate or constant.

Operand 3: Signed binary(4) variable scalar.

Bound program access

```
Built-in number for TESTPTR is 538.
TESTPTR (
         source_pointer   : pointer(16)
         test_type        : literal(1) OR
                            literal(4)
) :   signed binary(4)  /* return_code */
```

Description: Test the pointer specified by *source pointer* in the manner specified by *test type*. The value of the *return code* is determined by *test type*.

The *test type* operand may be declared as a literal of any scalar data type. If *test type* is a 4 byte literal, only the least significant byte is used to determine the operation to be performed. The remaining bytes must be binary zero.

Test Type	Description
Hex 00	Test the *source pointer* procedure pointer to see if it points to a procedure expecting *optimized procedure parameter passing*. If *source pointer* is a *null pointer value* then a *pointer does not exist* (hex 2401) exception is signalled. If *source pointer* is not a procedure pointer then a *pointer type invalid* (hex 2402) exception is signalled. If *source pointer* is a procedure pointer, but it identifies an activation which does not exist then an *object destroyed* (hex 2202) exception is signalled. When an exception is signalled, the value of *return code* is undefined. When an exception is not signalled, the value of *return code* will be: **0 =** *source pointer* does not point to a procedure expecting optimized parameter passing. **1 =** *source pointer* points to a procedure expecting optimized parameter passing.
Hex 01	Test the *source pointer* to see if it points to teraspace. If *source pointer* is a *null pointer value* then a *pointer does not exist* (hex 2401) exception is signalled. If *source pointer* is not a space pointer then a *pointer type invalid* (hex 2402) exception is signalled. When an exception is signalled, the value of *return code* is undefined. When an exception is not signalled, the value of *return code* will be: **0 =** *source pointer* does not point to teraspace. **1 =** *source pointer* points to teraspace.
Hex 02-FF	Reserved.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Test Subset (TESTSUBSET)

Bound program access
Built-in number for TESTSUBSET is 464. TESTSUBSET (first_source_string : address of aggregate(*) second_source_string : address of aggregate(*) string_length : unsigned binary(4,8) literal value which specifies the length of the two strings) : unsigned binary(4,8) value specifying the boolean result of the test /* result */

Description: Each byte value of the *first source string*, for the number of bytes indicated by *string length*, is logically **and**ed with the corresponding byte value of the *second source string*, on a bit-by-bit basis. The results are then compared with the *first source string*. The result is set to 1 if the strings are equal, otherwise it is set to 0. If the strings overlap in storage, predictable results occur only if the overlap is fully coincident.

If the space(s) indicated by the two addresses are not long enough to contain the number of bytes indicated by *string length*, a *space addressing violation* (hex 0601) exception is signalled. Partial results in this case are unpredictable.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Test Temporary Object (TESTTOBJ)

Op Code (Hex)	Extender	Operand 1	Operand 2 [3]
TESTTOBJB 1CA1	Branch options	Object	Branch targets
TESTTOBJI 18A1	Indicator options	Object	Indicator targets

Operand 1: System pointer.

Operand 2 [3]:

•

• *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.

• *Indicator Form*-Numeric variable scalar or character variable scalar.

Bound program access

```
Built-in number for TESTTOBJ is 462.
TESTTOBJ (
          object   : address of system pointer
) :   signed binary(4) /* return_code */
```

The return code will be set as follows:

Return code	Meaning
1	Pointer addresses temporary object
0	Pointer does not address temporary object (permanent object)

This built-in function is used to provide support for the branch and indicator forms of the TESTTOBJ operation. The user must specify code to process the *return code* and perform the desired branching or indicator setting.

Description: The object addressed by the pointer in operand 1 is checked to determine if it is a temporary object.

Note: A temporary object is an object that does not persist across IPL's. It is automatically destroyed at system termination.

Based on results, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to indicator operand (indicator form).

Operand 1

The *object* to be tested is addressed by this system pointer. Any system object can be tested. An unresolved system pointer will be resolved by this instruction.

Resultant Conditions:

-
- Temporary-The object specified by the system pointer is a temporary object.

  This is the equal condition.
- Not temporary-The object specified by the system pointer is not a temporary object. It is a permanent object.

  This is the unequal condition.

Warning: Temporary Level 3 Header

Authorization Required

-
- Object management
  -
  - Operand 1
- Execute

–
– Context referenced for address resolution

Lock Enforcement

•
• Materialize
  –
  – Operand 1
  – Context referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2204 Object Not Eligible for Operation

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Test User List Authority (TESTULA)

Op Code (Hex)
Extender
Operand 1
Operand 2
Operand 3
Operand [4-5]
TESTULA 10E7
Available authority template receiver
System object
Test options template
TESTULAB 1CE7
Branch options
Available authority template receiver
System object
Required authority template
Branch targets
TESTULAI 18E7
Indicator options
Available authority template receiver
System object
Required authority template
Indicator targets*Operand 1*: **Space pointer or null.**

Operand 2: **System pointer.**

Operand 3: **Space pointer.**

Operand 4-5:

•

• *Branch Form*-**Branch point, instruction pointer, relative instruction number, or absolute instruction number.**

• *Indicator Form*-**Numeric variable scalar or character variable scalar.**

Bound program access

```
Built-in number for TESTULA is 151.
TESTULA (
         available_authority_template_receiver   : address OR
                                                   null operand
         system_object                           : address of system pointer
         test_options_template                   : address
) :   signed binary(4) /* return_code */
```

The return code will be set as follows:

Return code	Meaning
1	Authorized.
0	Not Authorized.

This built-in function is used to provide support for the branch and indicator forms of the TESTULA instruction. The user must specify code to process the *return code* and perform the desired branching or indicator setting.

Description: **This instruction verifies that the object authorities and/or ownership rights specified by operand 3 are available to the user list specified by operand 3 for the object specified by operand 2. The user list consists of a governing user profile and a list of group profiles for a thread. The profiles can be specified either by system pointers, or a uid for the user profile and a gid for the group profiles. Any program adopted user profiles for the current thread are not included in the authority verification process when this instruction is used.**

If operand 1 is not null, all of the authorities and/or ownership specified by operand 3 that are currently available to the user list are returned in operand 1.

If the required authority is available, one of the following occurs:

•

• **Branch form indicated**

  –

  – **Conditional transfer of control to the instruction indicated by the appropriate branch target operand.**

• **Indicator form specified**

  –

  – **The leftmost byte of each of the indicator operands is assigned the following values:**

    - **Hex F1- If the result of the test matches the corresponding indicator option**

    - **Hex F0- If the result of the test does not match the corresponding indicator option**

If no branch options are specified, instruction execution proceeds to the next instruction.

The required authorities and/or ownership are specified by the *required authority* **field of operand 3. This field includes a test option that indicates whether all of the specified authorities are required or whether any one or more of the specified authorities is sufficient. This option can be used, for example, to test for operational authority by coding a template value of hex**

0F01 in operand 3. Using the *any* option does not affect what is returned in operand 1. If operand 1 is not null and the *any* option is specified, all of the authorities specified by operand 3 that are available to the user list are returned in operand 1.

If operand 1 is not null, it provides addressability to a template which contains a list of available authorization templates and has the following format:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided	Bin(4)
4	4		Number of bytes available	Bin(4)
8	8	Number of authorization templates returned	Bin(2)	
10	A	Reserved	Char(6)	
16	10	Authorization templates	[*] Char(2)	
16	10		Object control	Bit 0
16	10		Object management	Bit 1
16	10		Authorized pointer	Bit 2
16	10		Space authority	Bit 3
16	10		Retrieve	Bit 4
16	10		Insert	Bit 5
16	10		Delete	Bit 6
16	10		Update	Bit 7
16	10		Ownership (1 = yes)	Bit 8
16	10		Excluded	Bit 9
16	10		Authority list management	Bit 10
16	10		Execute	Bit 11
16	10		Alter	Bit 12
16	10		Reference	Bit 13
16	10		Reserved (binary 0)	Bits 14-15
*	*	—- End —-		

The first 4 bytes of the template identify the total number of bytes provided for use by the instruction. The value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length invalid* (hex 3803) exception to be signaled.

The second 4 bytes of the template identify the total number of bytes available to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length invalid* (hex 3803) exception) are signaled.

The number of authorization templates returned field is how many authorization templates were returned in the space provided. This does not indicate the total possible that could be returned. A maximum of one *authorization template* will be returned.

Operand 2 identifies the object for which authority is to be tested. The program adopted and propagated user profiles for any invocations of the current thread will not be included in the authority verification process of the user list.

Operand 3 identifies the profiles to be used in the authority verification process. The profiles or users may be indicated either by uid/gid or by system pointer, but not both in one request. A user profile (system pointer or uid) must always be specified and the number of group profiles (or gids) may be zero or more, but not negative. If the number of group profiles (or gids) is negative or greater than 17, the *template value invalid* (hex 3801) exception will be signaled. The format of the *test options template* (operand 3) is as follows:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Number of group profiles	Bin(2)
2	2	Required authority	Char(2)

Offset Dec	Hex	Field Name	Data Type and Length	
4	4	User indicator	Char(1)	
		Hex 80 = Users are identified by system pointers		
		Hex 40 = Profiles are identified by uid/gid		
5	5	Test flags	Char(1)	
5	5		Ignore pointer authority	Bit 0
5	5		Reserved	Bits 1-7
6	6	Reserved (binary 0)	Char(10)	
16	10	—- End —-		

The ignore pointer authority bit indicates whether the authority stored in the operand 2 system pointer should be ignored when testing the user list's authority to the object.

Immediately following this information will be the identification of the user profile and the group profile list. The first entry always identifies the user to be the user profile followed by a list of the users in the group profile list. The number of users in the list is indicated by the number of group profiles field. The format of the identification is either in system pointers or in uid/gids as specified by the user indicator field.

If the option for system pointers is selected, the user identification will have the following format.

Offset Dec	Hex	Field Name	Data Type and Length
0	0	User profile	System pointer
16	10	—- End —-	

Following the *user profile* pointer will be a list of system pointers for the group profiles. Each entry in the list will have the following format.

Offset Dec	Hex	Field Name	Data Type and Length
0	0	Group profile	System pointer
16	10	—- End —-	

If the option for uid/gid is selected, the user identification will have the following format.

Offset Dec	Hex	Field Name	Data Type and Length
0	0	User profile (uid)	UBin(4)
4	4	—- End —-	

Following the *user profile (uid)* will be a list of gids for the group profile list. Each entry in the list will have the following format.

Offset Dec	Hex	Field Name	Data Type and Length
0	0	Group profile (gid)	UBin(4)
4	4	—- End —-	

The format for the required authority field (operand 3) is as follows: (1 = authorized)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Authorization template	Char(2)	
0	0		Object control	Bit 0
0	0		Object management	Bit 1
0	0		Authorized pointer	Bit 2
0	0		Space authority	Bit 3
0	0		Retrieve	Bit 4
0	0		Insert	Bit 5
0	0		Delete	Bit 6
0	0		Update	Bit 7
0	0		Ownership (1 = yes)	Bit 8
0	0		Excluded	Bit 9
0	0		Authority list management	Bit 10
0	0		Execute	Bit 11
0	0		Alter	Bit 12
0	0		Reference	Bit 13
0	0		Reserved (binary 0)	Bit 14
0	0		Test option	Bit 15
			0 = All of the above authorities must be present.	
			1 = Any one or more of the above authorities must be present.	
2	2	—- End —-		

This instruction will tolerate a damaged object referenced by operand 2 when the reference is a resolved pointer. The instruction will not tolerate damaged contexts or programs when resolving pointers. Damaged user profiles encountered during the authority verification processing result in the signaling of the *authority verification terminated due to damaged object* **(hex 1005) exception.**

This instruction will not tolerate destroyed profiles or invalid system pointers/uid/gids to the users specified in operand 3. If system pointers are specified and any of them are null or do not point to a user profile, or if the uid or any of the gids are not in use or the user profile for it is destroyed, the *authority verification terminated due to destroyed object* **(hex 2207) exception will be signaled.**

Resultant Conditions:

•

- **Authorized - the required authority is available.**
- **Unauthorized - the required authority is not available.**

Warning: Temporary Level 3 Header

Authorization Required

•

- Execute
  –
  – Contexts referenced for address resolution

Lock Enforcement

•

- Materialize

–
– Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A0A ID Index Not Available

10 Damage Encountered

1002 Machine Context Damage State

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

3803 Materialization Length Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Transfer Control (XCTL)

Op Code (Hex)	Operand 1	Operand 2
0282	Program to be called or call template	Argument list

Operand 1: System pointer or space pointer data object.

Operand 2: Operand list or null.

Description: The instruction destroys the calling invocation and passes control to either the *program entry procedure* of a bound program or the external entry point of a non-bound program. If operand 1 specifies a Java(TM) program or a bound service program, an *invalid operation for program* (hex 2C15) exception is signaled.

Operand 1 may be specified as a system pointer which directly addresses the program that is to receive control or as a space pointer to a call template which identifies the program to receive control. Specifying a template allows for additional controls over how the specified program is to be invoked. The format of the *call template* is the following:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Call options	Char(4)	
0	0		Suppress adopted user profiles	Bit 0
			0 = No	
			1 = Yes	
0	0		Reserved (binary 0)	Bits 1-30
0	0		Force thread state to user state for transfer	Bit 31
			0 = No	
			1 = Yes	
4	4	Reserved (binary 0)	Char(12)	
16	10	Program to be called	System pointer	
32	20	—- End —-		

The **suppress adopted user profiles** option specifies whether or not the program adopted and propagated user profiles which may be serving as sources of authority to the thread are to be suppressed from supplying authority to the new invocation. Specifying *yes* causes the propagation of adopted user profiles to be stopped as of the calling invocation, thereby, not allowing the called invocation to benefit from their authority. Specifying *no* allows the normal propagation of adopted and propagated user profiles to occur. The called program may adopt its owning user profile, if necessary, to supplement the authority available to its invocation.

The **force thread state to user state for transfer** option specifies whether or not the transfer control needs to be done in the current thread state or change the thread state to user state.

Operand 2 specifies an operand list that identifies the arguments to be passed to the invocation to which control is being transferred. Automatic objects allocated by the transferring invocation are destroyed as a result of the transfer operation and, therefore, cannot be passed as arguments. An *argument list length violation* (hex 0802) exception is signaled if the number of arguments passed does not correspond to the number required by the parameter list of the target program.

An *unsupported space use* (hex 0607) exception is signalled if this call would pass a parameter stored in teraspace to a program which is not teraspace capable. To be teraspace capable, a non-bound program must be created as teraspace capable or a bound program must be created with a teraspace capable program entry procedure.

If the transferring invocation has an activation, the invocation count is decremented by 1.

If the transferring invocation has received control to process an exception, or an invocation exit, the *return instruction invalid* (hex 2C01) exception is signaled.

If the transferring invocation currently has an invocation exit set, the invocation exit is not given control and is implicitly cleared.

Common Program Call Processing: The details of processing differ for non-bound and bound programs. The following outlines the common steps.

1. A check is made to determine if the caller has authority to invoke the program and that the object is indeed a program object. The specified program must be either a bound program that contains a *program entry procedure* or a non-bound program.
2. The activation group in which the program is to be run is located or created if it doesn't exist.
3. If the program requires an activation entry and it is not already active within the appropriate activation group, it is activated. Bound programs always require an activation; non-bound programs require an activation only if they use static storage. The *invocation count* of a newly created activation is set to 1 while the *invocation count* of an existing activation is incremented by 1.
4. The invocation created for the target program has the following attributes (as would be reported via the Materialize Invocation Attributes (MATINVAT) instruction.)
   •
   - the *invocation mark* is at least one higher than any previous invocation within the thread. The *invocation mark* value is generated from the *thread mark counter* and is unique within the thread. There is no relationship between the values of the invocation mark and the marks of the *activation* or *activation group* associated with the invocation.
   - the *invocation number* is the same as the invocation number of the transferring invocation.
   - the *invocation type* is hex 02 to indicate a XCTL type of invocation.
5. The automatic storage frame (ASF), if required, is allocated on a 16-byte boundary.
6. Control is transferred to the program entry procedure (or external entry point) of the program.
7. Normal flow-of-control resumes at the instruction following the caller of the program issuing the XCTL instruction.

The details of locating the target activation group and activating the program differ depending upon the model of the program.

Bound Program: A bound program is activated and run in an activation group specified by program attributes. There are two logical steps involved:

•
- locate the existing, or create a new activation group for the program
- locate an existing, or create a new activation entry for the program within the activation group

After locating the activation entry for the program, control is passed to the program entry procedure for the program. If required, the activation group is destroyed when the invocation for the program entry procedure is destroyed.

Non-bound Program: The automatic storage frame begins with a 64 byte header. If the program defines no automatic data items the frame consists solely of the 64-byte header, otherwise the automatic storage items are located immediately following the header. In prior releases of the machine, this header contained invocation information which is now available via the Materialize Invocation Attributes (MATINVAT) instruction. This header is not initialized and the contents of the header are not used by the machine. (The space is allocated merely to provide for compatibility with prior implementations of the machine.) The *update PASA stack* program attribute, supported in prior implementations of the machine, is no longer meaningful and is ignored, if specified as an attribute of the program.

Warning: Temporary Level 3 Header

Authorization Required

•
- Execute
  –
  – Operand 1

– Contexts referenced for address resolution

Lock Enforcement

•

• Materialize

–

– Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0607 Unsupported Space Use

08 Argument/Parameter

0801 Parameter Reference Violation

0802 Argument List Length Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1C Machine-Dependent

1C02 Program Limitation Exceeded

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

2A Program Creation

2AB5 Observable Information Necessary For Retranslation Not Encapsulated

2C Program Execution

2C01 Return Instruction Invalid

2C15 Invalid Operation for Program

2C1D Automatic Storage Overflow

2C1E Activation Access Violation

2C1F Program Signature Violation

2C20 Static Storage Overflow

2C21 Program Import Invalid

2C22 Data Reference Invalid

2C23 Imported Object Invalid

2C24 Activation Group Export Conflict

2C25 Import Not Found

2C2A Caller Parameter Mask Does Not Match Imported Procedure Parameter Mask

2C2B Invalid Storage Model

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Transfer Object Lock (XFRLOCK)

Op Code (Hex)	Operand 1	Operand 2
0382	Receiving process control space	Lock transfer template

Operand 1: System pointer.

Operand 2: Space pointer.

Bound program access

```
Built-in number for XFRLOCK is 54.
XFRLOCK (
         receiving_process_control_space   : address of system pointer
         lock_transfer_template            : address
)
```

Description: Locks designated in the *lock transfer template* (operand 2) are either allocated to the receiving process (operand 1), the thread identified within the *lock transfer template*, or to the transaction control structure attached to the thread issuing this instruction. Upon completion of the transfer lock request, the current process, thread, or transaction control structure no longer holds the transferred lock(s).

Operand 2 identifies the objects and the associated lock states that are to be transferred to the receiving process, or to the receiving thread, or to the transaction control structure attached to the thread issuing this instruction. The space contains a system pointer to each object that is to have a lock transferred and a byte which defines whether this entry is active. If the entry is active, the space also contains the lock states to be transferred. Operand 2 must be aligned on a 16-byte boundary. The format is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Number of lock transfer requests in template	Bin(4)	
4	4	Offset to lock state selection bytes	Bin(2)	
6	6	Reserved	Char(8) +	
14	E	Transfer lock options	Char(2)	
14	E		Reserved	Bits 0-6 +
14	E		Template extension specified	Bit 7
			0 = Template extension is not specified.	
			1 = Template extension is specified.	
14	E		Lock scope	Bit 8
			0 = Transfer *lock scope object type* locks.	
			1 = Transfer thread scoped locks.	
14	E		Lock scope object type	Bit 9

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = Process containing the current thread.	
			1 = Transaction control structure attached to the current thread.	
14	E		Change lock scope	Bit 10
			0 = No.	
			1 = Yes.	
14	E		Reserved (binary 0)	Bits 11-15
16	10	—- End —-		

The *transfer lock template extension* is only present if *template extension specified* is indicated above. Otherwise, the *object lock(s) to be transferred* should immediately follow.

Offset Dec	Hex	Field Name	Data Type and Length
0	0	Transfer lock template extension	Char(16)
0	0		Extension options
0	0		Reserved
0	0		New lock scope
			0 = Lock scope is *lock scope object type*.
			1 = Lock scope is to the current thread.
0	0		Lock scope object type
			0 = Process containing the current thread.
			1 = Transaction control structure attached to the c
0	0		Reserved
1	1		Extension area
1	1		Reserved
9	9		Reserved (binary 0)
12	C		Open thread handle
16	10	Object lock(s) to be transferred	[*] System pointer
		(repeated as specified by *number of lock transfer requests in template* above)	
*	*	—- End —-	

The *lock state selection* is located by adding the *offset to lock state selection bytes* above to operand 2.

Offset Dec	Hex	Field Name	Data Type and Length	
0	0	Lock state selection	[*] Char(1)	
		(repeated for each pointer in the template)		
0	0		Lock state to transfer. Only one state may be requested per entry. (1 = transfer)	Bits 0-4
0	0		LSRD	Bit 0
0	0		LSRO	Bit 1
0	0		LSUP	Bit 2
0	0		LEAR	Bit 3
0	0		LENR	Bit 4
0	0		Reserved (binary 0)	Bit 5 +

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0		Lock count	Bit 6
			0 = The current lock count is transferred.	
			1 = A lock count of 1 is transferred.	
0	0		Entry active indicator	Bit 7
			0 = Entry not active. This entry is not used.	
			1 = Entry active. This lock is transferred.	
*	*	—- End —-		

Note: Fields indicated by a plus sign (+) are ignored by the instruction.

Locks to be transferred in one instruction must be either all process scoped, all thread scoped, or all transaction control structure scoped. Process scoped locks may be transferred from the process containing the current thread to the receiving process or to the transaction control structure that is attached to the current thread. Thread scoped locks may be transferred from the current thread to either the initial thread of the process associated with the *receiving process control space*, the thread specified by the **open thread handle** field in the *transfer lock template extension*, or to the transaction control structure attached to the current thread. Transaction control structure scoped locks may be transferred from the transaction control structure attached to the current thread to either the process containing the current thread or the current thread.

The **lock scope** for a lock held on an object may be changed. When **change lock scope** has a value of *no*, the following lock transfers are permitted:

•

- If *lock scope* specifies *transfer thread scoped locks* and no template extension is specified then the initial thread of the receiving process identified by operand 1 is used for the transfer.
- If *lock scope* specifies *transfer thread scoped locks* and a template extension is specified and the *open thread handle* is binary zero, then the initial thread of the receiving process identified by operand 1 is used for the transfer.
- If *lock scope* specifies *transfer thread scoped locks* and an *open thread handle* is specified, then the thread associated with the *open thread handle* is used for the transfer. The receiving process identified by operand 1 is ignored.
- If *lock scope* specifies *transfer lock scope object type locks* and the *lock scope object type* is *process containing the current thread*, then the receiving process identified by operand 1 is used for the transfer. The *open thread handle* is ignored.
- If *lock scope* specifies *transfer lock scope object type locks* and the *lock scope object type* is *transaction control structure attached to the current thread*, then a *template value invalid* (hex 3801) exception is signaled.

When *change lock scope* has a value of *no*, the *new lock scope* and *lock receiver object type* fields in the template extension are ignored.

When *change lock scope* has a value of *yes*, the template extension must be present. If the template extension is not present, a *template value invalid* (hex 3801) exception is signaled.

When change lock scope is specified, then locks may either be transferred between the transaction control structure attached to the current thread and the current thread or the process containing the current thread, the process containing the current thread and the transaction control structure attached to the current thread or the current thread, or the current thread and the process containing the current thread

or the transaction control structure attached to the current thread. Table 1 (page 1209) defines the permissible lock transfers when change lock scope is specified.

Table 1. Change Lock Scope

Current Lock Scope	New Lock Scope			
			Thread	
	Process[1]	Transaction control structure[2]	Process[3]	Transaction control structure[4]
Process[1]	Ignored	Allowed	Allowed	Allowed
Transaction control structure[2]	Allowed	Ignored	Allowed	Allowed
Thread/Process[3]	Allowed	Allowed	Ignored	Allowed
Thread/Transaction control structure[4]	Allowed	Allowed	Allowed	Ignored

Notes:

1 Lock scope is process containing the current thread.

2 Lock scope is transaction control structure attached to the current thread.

3 Lock scope is thread with the process containing the current thread as the logical parent for lock conflict checking.

4 Lock scope is thread with the transaction control structure attached to the current thread as the logical parent for lock conflict checking.

When *change lock scope* has a value of *yes*, the receiving process identified by operand 1 and the *open thread handle* are ignored.

If the receiving process control space (operand 1) does not reference an active process, the *process control space not associated with a process* (hex 2802) exception is signaled and no locks are transferred. If the receiving thread identified by the open thread handle does not reference an active thread, the *thread handle not associated with an active thread* (hex 2804) exception is signaled and no locks are transferred.

If either *lock scope object type* or *lock receiver object type* has a value of *transaction control structure attached to the current thread* and a transaction control structure is not attached to the current thread, an *object not available to process* (hex 2205) exception is signaled. If *lock receiver object type* has a value of *transaction control structure attached to the current thread* and the transaction control structure state does not allow objects to be locked on behalf of the transaction control structure, a *object not eligible for operation* (hex 2204) exception is signaled.

If a thread contained by the receiving process is issuing the instruction to transfer process scoped locks, then no operation is performed, and no exception is signaled. If the thread issuing the instruction is the receiver of the transferred thread scoped locks, then no operation is performed, and no exception is signaled. If the transaction control structure attached to the current thread is the receiver of transaction control structure scoped locks, then no operation is performed, and no exception is signaled.

The **lock count** transferred is either the current lock count held by the transferring process, thread or transaction control structure or a count of 1. If the receiving process, thread or transaction control structure already holds an identical lock, then the final lock count is the sum of the count originally held by the receiving process, thread or transaction control structure and the transferred count.

Only process scoped locks currently allocated to the process containing the thread issuing the instruction or thread scoped locks currently allocated to the thread issuing the instruction or the transaction control structure attached to the thread issuing the instruction can be transferred. If the transfer of an allocated lock would result in the violation of the lock allocation rules, then the lock cannot be transferred. An implicit lock may not be transferred.

No locks are transferred if an entry in the template is invalid.

The locks specified by operand 2 are transferred sequentially and individually. If one lock cannot be transferred because the process or thread does not hold the indicated lock on the object, then exception data is saved to identify the lock that could not be transferred. Processing of the next lock to be transferred continues.

If any lock was not transferred, the *invalid object lock transfer request* (hex 1A04) exception is signaled.

When an object lock is transferred, the transferring thread or its containing process or the transaction control structure attached to the thread synchronously loses the record of the lock, and the object is locked to the receiving process, thread or transaction control structure. However, the receiving thread obtains the lock asynchronously after the instruction currently being executed in that thread is completed. If the target is a process, the receiving process obtains the lock asynchronously after the instruction currently being executed in the initial thread of that process is completed. The receiving transaction control structure obtains the locks before the completion of this instruction. If the transferring process, thread or transaction control structure holds multiple locks for the object, any lock states not transferred are retained in the process, thread or transaction control structure.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

  –

  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize

  –

  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1A04 Invalid Object Lock Transfer Request

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2204 Object Not Eligible for Operation

2205 Object Not Available to Process

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

28 Process/Thread State

2802 Process Control Space Not Associated with a Process

2804 Thread Handle Not Associated with an Active Thread

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Translate (XLATE)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
1094	Receiver	Source	Position	Replacement

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar or null.

Operand 4: Character scalar.

Description: Selected characters in the string value of the *source* operand are translated into a different encoding and placed in the *receiver* operand. The characters selected for translation and the character values they are translated to are indicated by entries in the *position* and *replacement* strings. All the operands must be character strings. The *source* and *receiver* values must be of the same length. The *position* and *replacement* operands can differ in length. If operand 3 is null, a 256-character string is used, ranging in value from hex 00 to hex FF (EBCDIC collating sequence).

The operation begins with all the operands left-adjusted and proceeds character by character, from left to right until the character string value of the *receiver* operand is completed.

Each character of the *source* operand value is compared with the individual characters in the *position* operand. If a character of equal value does not exist in the *position* string, the *source* character is placed unchanged in the *receiver* operand. If a character of equal value is found in the *position* string, the corresponding character in the same relative location within the *replacement* string is placed in the *receiver* operand as the *source* character translated value. If the *replacement* string is shorter than the position string and a match of a source to *position* string character occurs for which there is no corresponding replacement character, the *source* character is placed unchanged in the *receiver* operand. If the *replacement* string is longer than the position string, the rightmost excess characters of the *replacement* string are not

used in the translation operation because they have no corresponding position string characters. If a character in the *position* string is duplicated, the first occurrence (leftmost) is used.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

The *receiver, source, position*, and *replacement* operands can be variable length substring compound operands.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for all of the operands on this instruction. The effect of specifying a null substring reference for either the *position* or *replacement* operands is that the *source* operand is copied to the *receiver* with no change in value. The effect of specifying a null substring reference for both the *receiver* and the *source* operands (they must have the same length) is that no result is set.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C08 Length Conformance

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Translate Bytes (XLATEB)

Bound program access
Built-in number for XLATEB is 24. XLATEB (source_string : address of aggregate(*) translate_table : address of aggregate(256) translate_length : unsigned binary(4) value which specifies the length of the source string to translate)

Description: Translates the data specified by *source string*. *Translate length* specifies the number of bytes to translate. Each byte of storage is modified with the corresponding entry in the translation table specified by *translate table*.

The translation table is exactly 256 bytes in length and specifies the translated values for the 256 possible byte values. The results are undefined if the table is less than 256 bytes.

The *translate table* and the *source string* should not overlap. Otherwise, the results are undefined.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

24 Pointer Specification

2401 Pointer Does Not Exist

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Translate Bytes One Byte at a Time (XLATEB1)

Bound program access

```
Built-in number for XLATEB1 is 414.
XLATEB1 (
         receiver           : address of aggregate(*) for the results of
                              the translation
         source_string      : address of the source bytes to translate
         translate_table    : address of aggregate(256)
         translate_length   : unsigned binary(4) value which specifies the
                              number of bytes to translate
)
```

Description: Translates the data specified by *source string* into the *receiver*. *Translate length* specifies the number of bytes to translate. Each byte of the *source string* is translated using the corresponding entry in the translation table specified by *translate table*.

Contrast this operation with XLATEB, which does not support overlapping operands.

Bytes are translated as follows:

•
- The source byte value is used as an offset and added to the location of *translate table*.
- The byte value contained in the offset location is the translated byte. This value is copied to the *receiver* in the same relative position as the original byte value within the *source string*.

The translation table is exactly 256 bytes in length and specifies the translated values for the 256 possible byte values. The results are undefined if the table is less than 256 bytes.

If *receiver* overlaps with *source string* and/or *translate table,* the overlapped operands are updated for every byte translated. The operation proceeds from left to right, one byte at a time. The following example shows the results of an overlapped operands translate operation. *Receiver, source string,* and *translate table* are coincident, with a value of hex 050403020103.

Hex 050403020103 - Initial value

Hex 030403020103 - After the 1st character is translated

Hex 030103020103 - After the 2nd character is translated

Hex 030102020103 - After the 3rd character is translated

Hex 030102020103 - After the 4th character is translated

Hex 030102020103 - After the 5th character is translated

Hex 030102020102 - After the 6th character, the final result

Warning: Temporary Level 3 Header

Authorization Required

•
- None

Lock Enforcement

•
- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Translate Multiple Bytes (XLATEMB)

Op Code (Hex)	Operand 1
1071	Translation template

Operand 1: Space pointer.

Bound program access
Built-in number for XLATEMB is 390. XLATEMB (translation_template : address)

Description: The source data string specified in the operand 1 *translation template* is converted starting with the left-most input byte using the *function byte*, *control map 1*, *control map 2*, and the *verification map*. The converted data string is returned in the *receiver* space specified in the operand 1 *translation template*.

Terminology:

ASCII Abbreviation for American National Standard Code for Information Interchange.

Control map A special layout of bytes used to control data conversion. The different types of *control maps* will be discussed later in this document.

Code page A collection of characters assigned to code points.

Code points A unique bit-pattern assigned to each graphic character, to be used by the computer when entering, storing, viewing, printing, or exchanging information.

Double Byte Character Set (DBCS) A set of characters in which each character is represented by a 2-byte code.

Endian The order of the bytes in memory. On *big endian* systems the most significant value is stored in the lowest address. On *little endian* systems the least significant value is stored in the lowest address. For example, take the integer value 13488. In big endian it is stored as hex '34B0' and on little endian it would be stored as hex 'B034'. The machine by default is *big endian*.

EBCDIC Abbreviation for extended binary coded decimal interchange code.

Graphic Term used to designate pure DBCS data.

ISO/IEC 10646 The international standard used to represent most of the world's written languages by assigning multiple bytes for each character. This standard was written by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC).

Mixed data Data that contains single-byte and double-byte encoding.

Well formed mixed data
Mixed data where DBCS data is bracketed by shift-out (SO) and shift-in (SI) controls.

Octet
An ordered sequence of eight bits, considered as a unit.

Single Byte Character Set (SBCS)
A set of characters in which each character is represented by a 1-byte code.

Substitution Value
A single-byte or multiple-byte code to be output from a conversion when the input character is not found in a *ward control block* or *ward*.

UCS-2 Level 1
Defines the form and level of *UCS*. *UCS-2* is a 16-bit form of *UCS*. Level 1 is an implementation level that does not support combining of characters. Every UCS-2 Level 1 character must be made up of only 2 bytes.

UTF-16
Defines a form and level of UCS. UTF-16 allows access to 63K characters as single UCS 16-bit units. It can access an additional 1 million characters by a mechanism known as surrogate pairs. Two ranges of UCS code values are reserved for the high (first) and low (second) values of these pairs. The high range is from hex D800 to hex DBFF and the low range is from hex DC00 to hex DFFF. A properly formed surrogate requires a high range code value followed by a low range value to form a valid character.

UTF-8
UTF-8 is the Unicode Transformation Format that serializes a Unicode code point as a sequence of one to four bytes. These sequences are mathematically equivalent to the set of UTF-16 characters.

Universal Multiple-Octet Coded Character Set (UCS)
Character set defined by IS0/IEC standard 10646.

Ward
A set of 256 single-byte or multiple-byte codes where all of the codes share a common first hex input byte when converting from a 2-byte code.

Ward control block
A set of 256 2-byte values within a *control map*. The 2-byte values provide the offsets from the start of the *control map* space to the beginning of all of the wards within the *control map*.

Operand 1:

Operand 1 is a space pointer to a 128-byte *translation template* aligned on a 16-byte boundary. If the *translation template* is not aligned on a 16-byte boundary, *boundary alignment* (hex 0602) exception is signaled.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Function	Bin(2)	
2	2	Control flags	Char(2)	
2	2		Control map type	Bit 0
			0 = *Control map* type D not supplied.	
			1 = *Control map* type D supplied.	
2	2		Substitution check	Bit 1
			0 = Do not check for *substitution*.	
			1 = Check for *substitution* on conversion from UCS.	
2	2		Override default multiple-byte substitution value	Bit 2
			0 = Use the default UCS-2 Level 1 *multiple-byte substitution value*.	
			1 = Use the specified *multiple-byte substitution value*.	
2	2		Ward transparency	Bit 3

Offset Dec	Offset Hex	Field Name	Data Type and Length	
			0 = Use the *multiple-byte substitution value* when converting characters in an empty ward.	
			1 = Do not convert characters in an empty ward.	
2	2		Well formed mixed data output	Bit 4
			0 = Do not ensure well formed mixed data on output	
			1 = Ensure well formed mixed data on output	
2	2		Caching requested	Bit 5
			0 = Do not request caching for *function* hex 0100 requests	
			1 = Do request caching for *function* hex 0100 requests	
2	2		Endian mode	Bit 6
			0 = Handle the input and output of Unicode data as *big endian*	
			1 = Handle the input and output of Unicode data as *little endian*	
2	2		UTF-16 casing map	Bit 7
			0 = Use *UCS-2 Level 1* maps for casing requests	
			1 = Use *UTF-16* maps for casing requests	
2	2		Reserved (binary 0)	Bits 8-15 +
4	4	Source length	Bin(4)	
8	8	Receiver buffer length	Bin(4)	
12	C	Receiver converted data length	Bin(4)	
16	10	Source range	Char(4)	
16	10		Range 1 lower limit	Bits 0-7
16	10		Range 1 upper limit	Bits 8-15
16	10		Range 2 lower limit	Bits 16-23
16	10		Range 2 upper limit	Bits 24-31
20	14	Single-byte substitution value	Char(1)	
21	15	Multiple-byte substitution value	Char(2)	
23	17	Reserved (binary 0)	Char(41) +	
64	40	Source	Space pointer	
80	50	Receiver	Space pointer	
96	60	Verification pointer	Space pointer	
112	70	Control map	Space pointer	
128	80	—- End —-		

Note: Fields annotated with a plus sign (+) are reserved fields. A reserved field value of non-zero results in the signaling of the *template value invalid* (hex 3801) exception.

Translation Template Field Descriptions:

Function

The *function* selected determines the type of conversion to be performed. Table 1 (page 1221) outlines the types of conversions that may be performed and the operands required for each *function*. The Table 1 (page 1221) columns are defined as follows:

-
- Function - The function selected.
- Control map type - The type of *control map* required as input for the given *function*.
- Verification map allowed - A *verification map* is allowed for the given *function*. A *verification map* is never required.
- Source data type - The *source* data type required for the given *function*.
- Receiver data type - The *receiver* data type returned for the given *function*.
- Estimated required buffer size - Value used to determine the actual required *receiver* buffer length. Multiply the *estimated required buffer size* value by the *source length* to get the required *receiver buffer length*. If this value is less than the minimum buffer size, use the minimum buffer size value for the *receiver buffer length*.

Table 1. XLATEMB supported functions

Function (hex)
Control map type

Verification map allowed
Source data type

Receiver data type
Estimated required buffer size

0001 A or D

No SBCS

UCS-2/UTF-16
2

0002 B or D

Yes UCS-2/UTF-16

SBCS .5

0003 C or D

No Graphic

UCS-2/UTF-16
1

0004 C or D

Yes UCS-2/UTF-16

Graphic 1

0005 C or D

No Mixed EBCDIC

UCS-2/UTF-16
2

0006 C or D

Yes UCS-2/UTF-16

Mixed EBCDIC
2

0007 C

No Mixed ASCII

UCS-2/UTF-16
2

Control flags

-

- **Control map type:** Determines which control map type will be supplied for the specified *function*. This field is verified against the *function* specified when a *type Dcontrol map* is specified. A control map type of D specified with an incorrect function results in the signaling of the *template value invalid* (hex 3801) exception. Refer to Table 1 (page 1221) for details on which *functions* require this flag to be set.
- **Substitution check:** Check for *substitution* on conversion from UCS-2 Level 1 or UTF-16 or UTF-8 data. *Substitution check* is only supported for *functions* 0002, 0004, 0006, 0008, 000C, 0034, 0036, and 0038.

  When a *substitution* character is encountered, a *substitution character used* (hex 0C20) exception is signaled at instruction completion. Complete results are placed in the *receiver* and *receiver converted data length* fields. The number of substitutions in the *receiver* data will be stored in the *number of substitutions* field of the exception data for the *substitution character used* (hex 0C20) exception.
- **Override default multiple-byte substitution value:** Determines which *multiple-byte substitution value* will be placed into the *receiver* space when using a *type C* or *type D control map* and substitution is required on conversion to UCS-2 Level 1 data. *Override default multiple-byte substitution value* is only supported for functions 0003, 0005, and 0007.

  When substitution is required and the function is 0003, 0005, or 0007, the *multiple-byte substitution value* will be one of the following:

  –

  – If the *override default multiple-byte substitution value* is 0, the default UCS-2 Level 1 substitution value of hex FFFD will be used.

  – If the *override default multiple-byte substitution value* is 1, the *multiple-byte substitution value* specified in the template will be used.

  Note: The *override default multiple-byte substitution value* field must be set to 0 for functions 0001, 0002, 0004, 0006, 0008, 0009, 000A, 000B, 000C, 0033, 0034, 0035, 0036, 0037, 0038, 003B, 003C, 00FE, 00FF and 0100 or a *template value invalid* (hex 3801) exception will be signaled.
- **Ward transparency:** Determines whether *source* characters in an empty ward are converted using a *multiple-byte substitution value*, or moved to the *receiver* space transparently with no conversion taking place. *Ward transparency* is only supported for *functions* 0003, 0004, 0005, and 0007. If the *ward transparency* field is binary 1, then the *multiple-byte substitution value* is not used and the *override default multiple-byte substitution value* field is ignored.

  Note: The *ward transparency* field must be set to binary 0 for functions 0001, 0002, 0006, 0008, 0009, 000A, 000B, 000C,0033, 0034, 0035, 0036, 0037, 0038, 003B, 003C, 00FE, 00FF and 0100 or a *template value invalid* (hex 3801) exception will be signaled. When using the *ward transparency* feature with *functions* 0005 and 0007, the first entry in the *control map ward control block* must be non-zero, or a *template value invalid* (hex 3801) exception will be signaled.
- **Well formed mixed data output:** Determines whether a DBCS character in the last two bytes of the *receiver* space (as defined by *receiver buffer size*) should be replaced by a shift-in control.

  Note: The *well formed mixed data output* field must be set to 0 for function 0100 or a *template value invalid* (hex 3801) exception will be signaled.

  Note: This control is only applied where data truncation is necessary. Mixed data output will always be well formed if the *receiver* space is large enough.

Source length

The length of the source data contained in the space addressed by the *source* space pointer. A length value of less than 1 results in the signaling of the *template value invalid* (hex 3801) exception.

Receiver buffer length

The length of the receiver space pointed to by the receiver space pointer. A length value of less than 1 results in the signaling of the *template value invalid* (hex 3801) exception.

Receiver converted data length

The length of the data placed in the *receiver* space after conversion. This field is set by the machine and will always be less than or equal to the value specified for *receiver buffer length*.

Source range

The range of the double-byte content of the mixed ASCII *source* input data. *Source range* is only used with *function* 0007. The *source range* field is divided into 2 ranges, range 1 and range 2. Each range has a 1 byte lower and 1 byte upper limit. Some actual working examples of *source ranges* are defined below:

Source Ranges
Supported Language

Hex 819FE0FC
Japanese

Hex 81BF0000
Korean

Hex 81FC0000
Simplified Chinese

Hex 81FC0000
Traditional Chinese

Hex 8FFE0000
Republic of Korea National Standard

Note: A *template value invalid* (hex 3801) exception will be signaled if one of the following occurs:

-
- If the *function* is 0007 and range 1 is set to nulls.
- If the *function* is 0007 and the upper limit is less than the lower limit for either *range 1* or *range 2*.

If range 2 is nulls then it will not be used and no exception will be signaled.

Note: The *source range* field must be set to binary 0 for any function except *function* 0007 or a *template value invalid* (hex 3801) exception will be signaled.

Single-byte substitution value

A single-byte value to be output from a conversion when the following occurs:

-
- A *type C* or *type D control map ward control block* entry is hex zeros and single-byte data is being processed.
- A *type B control mapward control block* entry is hex zeros.

Note: The *single-byte substitution value* must be set to hex 00 for functions 0001, 0003, 0004, 0005, 0007, 0009, 000A, 000B, 000C, 0033, 0035, 0036, 0037, 003B, 003C, 00FE, 00FF and 0100 or a *template value invalid* (hex 3801) exception will be signaled.

Multiple-byte substitution value

A 2-byte value to be output from a conversion when a *type C or type D control map ward control block* entry is hex zeros and multiple-byte data is being processed. The *multiple-byte substitution value* is ignored if the *ward transparency* feature is being used.

Note: The *multiple-byte substitution value* must be set to hex 0000 for functions 0001, 0002, 0009, 000A, 000B, 000C, 0033, 0034, 0035, 0037, 003B, 003C, 00FE, 00FF and 0100 or a *template value invalid* (hex 3801) exception will be signaled. If the *function* is 0003, 0005, or 0007, and the *override default multiple-byte substitution value* field is 0, then the *multiple-byte substitution value* must be set to hex 0000 or a *template value invalid* (hex 3801) exception will be signaled.

Source

A space pointer to the source data. The length is defined by the *source length* field.

Receiver

A space pointer to the receiver data buffer. The number of bytes available is specified by the *receiver buffer length* field. If an error occurs during conversion this buffer will contain the data converted up to the point of the error. The length of the data converted is stored in the *receiver converted data length* field.

Note: Undefined results can occur if the storage locations specified by *source* and *receiver* overlap.

Verification pointer (optional)

A space pointer to a *verification map* to be used to verify UCS-2 Level 1 *source* data. The *verification map* has the following format:

Offset	
Dec	Hex
	Field Name
	Data Type and Length
0	0
	Map size
	UBin(2)
2	2
	Verification map entry
	Char(2)
4	4
	—- End —-

The *map size* is a hex value that indicates the number of 2-byte *verification map* entries that exist in the map.

Each *verification map entry* contains one UCS-2 Level 1 code.

Note: The number of 2-byte *verification map* entries is determined by the number specified in the *map size* field.

If a *verification map* is specified, it is used to verify that the UCS-2 Level 1 data in the *source* input is correct. The verification map contains a list of valid UCS-2 Level 1 codes. The map values must be encoded in *UCS-2 Level 1* and must be sorted in ascending numerical order. Failure to sort the *verification map* will result in unpredictable results. Refer to Table 1 (page 1221) for specific *function* codes which support use of the *verification map*. If any UCS-2 Level 1 code is not found in the *verification map* during the conversion, a *source verification error* (hex 0C21) exception is signaled. If unused, the *verification pointer* must be a null pointer value. This ensures no verification takes place.

Note: The *verification pointer* field must be set to a null pointer value for functions 0033, 0034, 0035, 0036, 0037, 0038, 003B, 003C, 00FE, 00FF and 0100 or a *template value invalid* (hex 3801) exception will be signaled.

The following is an example of a *verification map*:

```
0011009A01000101010201030104010501060107010801090 10A010B010C010D03B103B2
```

Table 2 (page 1225) shows the layout of the example *verification map* shown above with offsets and entry number included for clarity. The first 2-byte value, at offset hex 0000, indicates the number of 2-byte UCS-2 Level 1 codes in the remainder of the map. In this example, the first value in the map is hex 0011 (decimal 17), indicating that there are 17 2-byte codes in the remainder of the map.

Table 2. Verification map layout

Offset	Entry Number
Verification Map Value	
0000	
0011 (Number of entries in verification map.)	
0002	1
009A	
0004	2
0100	

Control map

A space pointer to a *control map* to be used in the conversion of the *source* data. Refer to Table 1 (page 1221) for information on types of maps required for the various *functions*. If unused, the *control map* pointer must be a null pointer value.

Control map types

The following list explains the different types of control maps:

Type A — Used to map SBCS data to *UCS-2 Level 1* multiple-byte data. The *type A control map* has the following format:

Offset

Dec	Hex
	Field Name
	Data Type and Length
0	0
	Type A control map
	[256] Char(2)
	A *type A control map* consists of 256 2-byte codes.
0	0
	Type A map entries
	Char(2)
	The *type A control map* entries are indexed by the hex input byte. Hex input values can range from 00 to FF for a total of 256 2-byte values. All entries are required to be fully populated for the index range of hex 00 to hex FF. To ensure proper conversion, unused entries should be set to some value.
512	200
	—- End —-

Note: The *type A control map* entry field is repeated 256 times to give the byte total of 512.

The following is a partial example of a *type A control map*:

```
0100010101020103010401050106010701080109010A010B010C...01FE01FF
```

Table 3 (page 1227) shows a partial layout of the example *type A control map* shown above with hex input values and offsets included for clarity. To find the result with a 3-byte input value of hex 0C0805, do the following:

1. Use the first input byte, hex 0C, to index into the map.
2. At offset hex 0018, the control map value is hex 010C.
3. Now index into the space, using the second input byte of hex 08.
4. At offset hex 0010, the control map value is hex 0108.
5. Now index into the space, using the third input byte of hex 05.
6. At offset hex 000A, the control map value is hex 0105.
7. The result placed in the receiver data buffer is hex 010C01080105.

Table 3. Type A map layout

Hex Input Value Offset	**Control Map Value**
00	0000

Template Value Invalid exception reason codes: This instruction supports setting of the optional reason code field in the exception data which can be retrieved when the *template value invalid* (hex 3801) exception is signaled. The *template value invalid reason codes* are defined as follows:

Reason Code	Description
Hex 0001	Template value is not valid. The template field in error can be determined by using the offset, stored in the template offset information of the exception data for the *template value invalid* (hex 3801) exception, to offset from the start of the operand 1 *translation template* to the start of the field in error.
Hex 0002	Unsupported function selected. No conversion will occur.
Hex 0004	The specified *type D control map* is not supported. No conversion will occur.
Hex 0005	The *source range* field was specified incorrectly for one of the following conditions: • • Range 1 is set to nulls. • The upper limit is less than the lower limit for either *range 1* or *range 2*. **Note:** Reason code hex 0005 can only occur when *function* 0007 is selected.
Hex 0006	The specified *type E control map* is invalid. No conversion will occur.

XLATEMB Examples:

Example 1: Convert hex 0B05 using XLATEMB *function 01* (convert from SBCS to UCS-2 Level 1). This example uses the *type A control map* in Table 3 (page 1227). To find the result:

1. Use the first byte of the input data, hex 0B, to index (2-bytes for each index value) into the *type A* map.
2. At offset hex 0016, the corresponding UCS-2 Level 1 value is hex 010B.
3. Use the second byte of the input data, hex 05, to index (2-bytes for each index value) into the *type A* map.
4. At offset hex 000A, the corresponding UCS-2 Level 1 value hex 0105 is output.
5. The instruction completes with a value of hex 010B0105 placed in the *receiver* and a hex 0004 will be placed in the *receiver converted data length*.

Example 2: Convert hex 03B1009A using *function* 0002 (convert from UCS-2 Level 1 to SBCS). This example uses the *verification map* in Table 2 (page 1225) and*type B* map in Table 4 (page 1227). To find the result:

1. The first UCS-2 Level 1 input value, hex 03B1, is compared against the *verification map*. Since the value is found at offset hex 0020 in the *verification map*, processing will continue.
2. Use the first byte of the input data, hex 03, to index into the *type B control map ward control block* starting at offset hex 0000.
3. At offset hex 0006, the *ward control block* entry value is hex 0300.
4. Use hex 0300 to offset from the start of the *control map* to the start of the *ward detail for ward 03*.
5. Use the second input byte, hex B1, to index into the *ward detail for ward 03*.
6. At offset hex 03B1, the corresponding SBCS value hex 8A is output.
7. The second UCS-2 Level 1 input value, hex 009A, will be compared against the *verification map*. Since the value is found at offset hex 0002 in the *verification map*, processing will continue.
8. Use the first byte of the input data, hex 00, to index into the *type B control map ward control block*, starting at offset hex 0000.
9. At offset hex 0000, the *ward control block* entry value is hex 0200.
10. Use hex 0200 to offset from the start of the *control map* to the start of the *ward detail for ward 00*.
11. Use the second input byte, hex 9A, to index into the *ward detail for ward 00*.

12. At offset hex 029A, the corresponding SBCS value hex 3A is output.
13. The final output of hex 8A3A is placed in the *receiver* and a hex 0002 will be placed in the *receiver converted data length*.

Warning: Temporary Level 3 Header

Authorization Required

•

• As appropriate for the space objects pointed to by the operand 1 template.

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

0A Authorization

0A01 Unauthorized for Operation

0C Computation

0C20 Substitution Character Used

0C21 Source Verification Error

0C22 Unpaired Shift Control

0C23 Source Information Error

0C24 Receiver Buffer Length Exceeded

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2403 Pointer Addressing Invalid Object Type

32 Scalar Specification

3203 Scalar Value Invalid

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Translate with Table (XLATEWT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
109F	Receiver	Source	Table

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Description: The *source* characters are translated under control of the translate *table* and placed in the *receiver*. The operation begins with the leftmost character of operand 2 and proceeds character-by-character, left-to-right.

Characters are translated as follows:

-
- The *source* character is used as an offset and added to the location of operand 3.
- The character contained in the offset location is the translated character. This character is copied to the *receiver* in the same relative position as the original character in the *source* string.

If operand 3 is less than 256 bytes long, the character in the *source* may specify an offset beyond the end of operand 3. If operand 2 is longer than operand 1, then only the leftmost portion of operand 2, equal to the length of operand 1, is translated. If operand 2 is shorter than operand 1, then only the leftmost portion of operand 1, equal to the length of operand 2, is changed. The remaining portion of operand 1 is unchanged.

If operand 1 overlaps with operand 2 and/or 3, the overlapped operands are updated for every character translated. The operation proceeds from left to right, one character at a time. The following example shows the results of an overlapped operands translate operation. Operands 1, 2, and 3 have the same coincident character string with a value of hex 050403020103.

Hex 050403020103-Initial value

Hex 030403020103-After the 1st character is translated

Hex 030103020103-After the 2nd character is translated

Hex 030102020103-After the 3rd character is translated

Hex 030102020103-After the 4th character is translated

Hex 030102020103-After the 5th character is translated

Hex 030102020102-After the 6th character, the final result

Note that the instruction does not use the length specified for the *table* operand to constrain access of the bytes addressed by the *table* operand.

If operand 3 is less than 256 characters long, and a *source* character specifies an offset beyond the end of operand 3, the result characters are obtained from byte locations in the space following operand 3. If that portion of the space is not currently allocated, a *space addressing violation* (hex 0601) exception is signaled. If operand 3 is a constant with a length less than 256, source characters specifying offsets greater than or equal to the length of the constant are translated into unpredictable characters.

All of the operands support variable length substring compound scalars.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for all of the operands on this instruction. Specifying a null substring reference for the *table* operand does not affect the operation of the instruction. In this case, the bytes addressed by the *table* operand are still accessed as described above. This is due to the definition of the function of this instruction which does not use the length specified for the *table* operand to constrain access of the bytes addressed by the *table* operand. The effect of specifying a null substring reference for either or both of the *receiver* and the *source* operands is that no result is set.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1044 Partial System Object Damage

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 2203 Object Suspended
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

- 3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Translate with Table and DBCS Skip (XLATWTDS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
1077	Target	Length	Table

Operand 1: Character variable scalar.

Operand 2: Binary(4) scalar.

Operand 3: Character scalar.

Bound program access

```
Built-in number for XLATWTDS is 148.
XLATWTDS (
          target   : address
          length   : address of unsigned binary(4)
          table    : address
)
```

Description: The simple (single byte) characters in the *target* are translated under control of the translate *table*, for the length defined by operand 2. The extended (double byte) character portions of the *target* are bypassed and not translated. The operation begins with the leftmost character of operand 1 and proceeds character-by-character, left-to-right, skipping over any Double byte character (DBCS) data portions.

The *target*, operand 1, should have double byte character data surrounded by a shift out control character (SO = hex 0E) and a shift in control character (SI= hex 0F). Once a SO character is encountered, the translating of single byte characters halts. The operation will then proceed double byte character-by-double byte character until a SI character is encountered. This shift in character is then used to restart the translating of single byte characters.

The *length* operand, operand 2, is the number of bytes and must contain a value between 1 and 32,767. For length values outside this range a *scalar value invalid* (hex 3203) exception is signaled.

Single byte characters are translated as follows:

-
- The *target* character is used as an offset and added to the location of operand 3.
- The character contained at the offset location of operand 3 is the translated character. This character replaces the original character in the *target*.

The following example shows the step-by-step results of this translate operation. The translate *table* for this example has the following hex value: C3D406C5D504C1C2C4C5C6C7C8C9C1C6

Hex 05040ED2D2E1E10F03 - Initial *target* value

Hex 04040ED2D2E1E10F03 - After the 1st character is translated

Hex 04D50ED2D2E1E10F03 - After the 2nd character is translated

Hex 04D50ED2D2E1E10F03 - SO character encountered, skip the DBCS portion

Hex 04D50ED2D2E1E10F03 - Resume translating after SI control character

Hex 04D50ED2D2E1E10FC5 - Translate 9th character

Hex 04D50ED2D2E1E10FC5 - Final *target* value

The translate *table*, operand 3, is assumed to be 256 bytes long. If the table is less than 256 characters long, and a *target* character specifies an offset beyond the end of the table, the resultant characters are obtained from byte locations in the space following translate *table*. If that portion of the space is not currently allocated, a *space addressing violation* (hex 0601) exception is signaled.

This operation only translates the *target* string and does not validate the double byte portions of the *target*. For example, if a DBCS portion of the target string is preceded by the Shift Out control character, but missing the closing Shift In character, then an *invalid extended character data* (hex 0C12) exception will NOT be signaled. However, the Copy Extended Characters Left-Adjusted With Pad (CPYECLAP) instruction can be used to validate extended character data, if necessary.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Trim Length (TRIML)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10A7	Receiver length	Source string	Trim character

Operand 1: Numeric variable scalar.

Operand 2: Character scalar.

Operand 3: Character(1) scalar.

Description: The operation determines the resultant length of operand 2 after the character specified by operand 3 has been trimmed from the end of operand 2. The resulting length is stored in operand 1. Operand 2 is trimmed from the end as follows: if the rightmost (last) character of operand 2 is equal to

the character specified by operand 3, the length of the trimmed operand 2 string is reduced by 1. This operation continues until the rightmost character is no longer equal to operand 3 or the trimmed length is zero. If operand 3 is longer than one character, only the first (leftmost) character is used as the trim character.

Operands 2 and 3 are not changed by this instruction.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0C Computation

0C0A Size

10 Damage Encountered

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Unlock Object (UNLOCK)

Op Code (Hex)	Operand 1
03F1	Unlock template

Operand 1: Space pointer.

Bound program access
Built-in number for UNLOCK is 55. UNLOCK (unlock_template : address)

Description: The instruction releases the object locks that are specified in the *unlock template*. The template specified by operand 1 identifies the system objects and the lock states (on those objects) that are to be released. The *unlock template* must be aligned on a 16-byte boundary. The format is as follows:

Offset			
Dec	Hex	Field Name	Data Type and Length
0	0	Number of unlock requests in template	Bin(4)
4	4	Offset to lock state selection bytes	Bin(2)
6	6	Reserved	Char(8) +
14	E	Unlock options	Char(2)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
14	E		Reserved	Bits 0-3 +
14	E		Unlock type	Bits 4-5
			00 = Unlock specific locks now allocated to process, thread, or transaction control structure	
			01 = Cancel specific asynchronously waiting lock request for the current thread, or release allocated locks for the current thread or its containing process	
			10 = Cancel all asynchronously waiting lock requests for current thread	
			11 = Invalid	
14	E		Reserved (binary 0)	Bits 6-7
14	E		Lock scope	Bit 8
			0 = Lock is scoped to the *lock scope object type*.	
			1 = Lock is scoped to the current thread.	
14	E		Lock scope object type	Bit 9
			0 = Process containing the current thread.	
			1 = Transaction control structure attached to the current thread.	
14	E		Reserved (binary 0)	Bits 10-15
16	10	Object(s) to unlock (one for each unlock request)	[*] System pointer	
*	*	—- End —-		

The *unlock options* is located by adding the **offset to lock state selection bytes** above to operand 1.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Unlock options (repeated for unlock request)	[*] Char(1)	
0	0		Lock state to unlock (only one state can be selected) (1 = unlock)	Bits 0-4
0	0		LSRD	Bit 0
0	0		LSRO	Bit 1
0	0		LSUP	Bit 2
0	0		LEAR	Bit 3
0	0		LENR	Bit 4
0	0		Lock count option	Bit 5
			0 = Lock count reduced by 1	
			1 = All locks are unlocked. The lock count is set to 0.	
0	0		Reserved (binary 0)	Bit 6 +
0	0		Entry active indicators	Bit 7

Offset Dec	Hex	Field Name	Data Type and Length
			0 = Entry not active. This entry is not used.
			1 = Entry active. These locks are unlocked.
*	*	—- End —-	

Note: Fields indicated by a plus sign (+) are ignored by the instruction.

The **unlock type** field specifies if locks are to be released or outstanding lock requests are to be canceled.

If all *asynchronous lock waits* are being canceled (*unlock type* specified as 10), then *objects to unlock* and *unlock options* for each object are not required. If the asynchronous lock fields are provided in the template, then the data is ignored.

Specifying 01 for *unlock type* attempts to cancel an asynchronous lock request that is identical to the one defined in the template. After the instruction attempts to cancel the specified request, program execution continues just as if 00 had been specified for *unlock type*. A waiting lock request is canceled if the number of active requests in the template, the objects, the objects corresponding lock states, and the order of the active entries in the template all match.

When a lock is released, the lock count is reduced by 1 or set to 0 in the specified state. This option is specified by the **lock count option** parameter.

If 01 is specified for *unlock type* and the *lock count option* for an object lock is 0 (lock count reduced by 1), then a successful cancel satisfies this request, and no additional locks on the object are unlocked. If the *lock count option* for an object lock is set to 1 (*set lock count to 0*), the results of the cancel are disregarded, and all held locks on the object are unlocked.

Specific locks can be unlocked only if they are allocated to the process, thread, or the transaction control structure attached to the thread issuing the unlock instruction. The lock scope specified by *lock scope* and *lock scope object type* must also match the scope of the locks currently allocated for the process, thread, or the transaction control structure. If *lock scope object type* has a value of *transaction control structure attached to the current thread* and a transaction control structure is not attached to the current thread, the lock must be allocated to the process containing the current thread.

Implicit locks may not be unlocked with this instruction. No locks are unlocked if an entry in the template is invalid.

Object locks to unlock are processed sequentially and individually. If one specific object lock cannot be unlocked because the process, thread, or the transaction control structure does not hold the indicated lock on the object, then exception data is saved, but processing of the instruction continues.

After all requested object locks have been processed, the *invalid unlock request* (hex 1A03) exception is signaled if any object lock was not unlocked.

If 01 is specified for *unlock type* and the cancel attempt is unsuccessful, an *invalid unlock request* (hex 1A03) exception is signaled when any object lock in the template is not unlocked.

Warning: Temporary Level 3 Header

Authorization Required

•

• Execute

  –

  – Contexts referenced for address resolution

Lock Enforcement

•

• Materialize

  –

  – Contexts referenced for address resolution

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

0A Authorization

0A01 Unauthorized for Operation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

1A Lock State

1A01 Invalid Lock State

1A03 Invalid Unlock Request

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2205 Object Not Available to Process

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Unlock Object Location (UNLOCKOL)

Op Code (Hex)	Operand 1
03C5	Unlock template

Operand 1: Space pointer.

Bound program access

```
Built-in number for UNLOCKOL is 499.
UNLOCKOL (
          unlock_template   : address
)
```

Description: The lock states specified in the *unlock template* (operand 1) are removed for the object locations specified in the *unlock template*.

Any object location(s) within the *unlock template* need not exist when this instruction is issued although the object pointer must be a valid pointer as used to lock the object location.

The *unlock template* identified by operand 1 must be aligned on a 16-byte boundary. The format of the *unlock template* is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Number of object location unlock requests in template	UBin(4)
4	4	Offset to unlock options	UBin(4)
8	8	Reserved (binary 0)	Char(24) +
32	20	Object location(s) to be unlocked	[*] Object pointer
		(repeated as specified by the *number of object location unlock requests in template* field above)	
*	*	—- End —-	

The *unlock options* field is located by adding the *offset to unlock options* field above to operand 1.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Unlock options	[*] Char(1)	
		(repeated for each unlock request)		
0	0		Lock state to unlock	Bits 0-4
			(1 = unlock requested, 0 = unlock not requested) Only one state may be requested; else the *template value invalid* (hex 3801) exception is signaled.	
0	0		LSRD lock	Bit 0
0	0		LSRO lock	Bit 1
0	0		LSUP lock	Bit 2
0	0		LEAR lock	Bit 3
0	0		LENR lock	Bit 4
0	0		Lock count option	Bit 5
			0 = Lock count reduced by 1	
			1 = All locks are unlocked. (The lock count is set to 0).	
0	0		Reserved (binary 0)	Bit 6
0	0		Entry active indicator	Bit 7
			0 = Entry not active. This entry is not used.	
			1 = Entry active. Lock is to be unlocked.	
*	*	—- End —-		

Note: Fields indicated with a plus sign (+) are ignored by the instruction.

This instruction can request the deallocation of one or more lock states on one or more object locations. The locks are deallocated sequentially until all specified locks are deallocated. When a lock is deallocated, the lock count is either reduced by 1 or set to 0 for the specified state. This option is specified by the **lock count option**.

Specific locks can be unlocked only if they are held by the thread issuing the unlock instruction. If an object location lock cannot be unlocked because the thread does not hold the indicated lock, then exception data is saved but processing of the instruction continues. After all requested object location

locks have been processed, the *invalid unlock request* (hex 1A03) exception is signaled if any requested object location lock was not unlocked because the thread did not have a lock on the object location.

No locks are unlocked if a template value is invalid.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1A Lock State

1A03 Invalid Unlock Request

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C06 Machine Lock Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

Unlock Pointer-Based Mutex (UNLKMTX)

Op Code (Hex)	Operand 1	Operand 2
03D6	Mutex	Result

Operand 1: Space pointer.

Operand 2: Signed binary(4) variable scalar.

Bound program access

```
Built-in number for UNLKMTX is 158.
UNLKMTX (
         mutex   : address
) :   signed binary(4)   /* result */
```

Note: The term ″mutex″ in this instruction refers to a ″pointer-based mutex″.

Description: The *mutex*, whose address is contained in operand 1, is released (unlocked).

The *mutex* must be aligned on a 16-byte boundary.

The *mutex* must have been previously created by the CRTMTX instruction or be a copy of a mutex that was previously created by the CRTMTX instruction, and must be currently allocated to the issuer by

means of a successful LOCKMTX instruction. See the CRTMTX instruction for additional information regarding mutex copies. An EPERM error number is returned if the *mutex* is not locked by the requesting thread.

Result is used to indicate the success or failure of the UNLKMTX instruction. If a non-recursive *mutex* is unlocked by this instruction, then *result* is set to 0. If the *mutex* has been recursively locked, this instruction will release one lock and *result* is set to a negative number whose absolute value is the number of locks which still remain on the *mutex*, if not 0. See the LOCKMTX instruction for additional information on using the recursive behavior of a mutex.

If an error occurs, then the *result* is set to an error number.

An EINVAL error number is returned if the *mutex* has not been initialized, or if it has been altered. The ETYPE error number is returned if the *mutex* operand references a synchronization object that is not a pointer-based mutex.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Error conditions

The *result* is set to one of the following:

EINVAL 3021 - The value specified for the argument is not correct.

EPERM 3027 - Operation not permitted.

ETYPE 3493 - Object type mismatch.

A synchronization object at this address is not a pointer-based mutex.

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Unlock Space Location (UNLOCKSL)

Op Code (Hex)	Operand 1	Operand 2
03F2	Space location or unlock template	Lock request

Operand 1: Space pointer data object.

Operand 2: Character(1) scalar or null.

Bound program access

```
Built-in number for UNLOCKSL is 56.
UNLOCKSL (
          space_location   : address of space pointer(16)
          lock_request     : address OR
                             null operand
)
```

Description: When operand 2 is not null, the lock type specified by operand 2 is removed from the *space location* (operand 1). When the operand 2 is null, the lock type is removed for the space locations specified in the *unlock template* (operand 1).

Any space location(s) specified by operand 1, or within the template specified by operand 1, need not exist when this instruction is issued although the space pointer must be a valid pointer as used to lock the space location.

A space pointer machine object cannot be specified for operand 1.

The following is the format of operand 2 when not null:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Lock request	Char(1)	
0	0		Lock state selection (1 = lock requested, 0 = lock not requested) Only one state may be requested per entry.	Bits 0-4
0	0		LSRD lock	Bit 0
0	0		LSRO lock	Bit 1
0	0		LSUP lock	Bit 2
0	0		LEAR lock	Bit 3
0	0		LENR lock	Bit 4
0	0		Reserved (binary 0)	Bits 5-7
1	1	—- End —-		

If a space location lock cannot be unlocked because the thread does not hold the indicated lock, then the *invalid space location unlocked* (hex 1A05) exception is signaled.

When operand 2 is null, the lock request template identified by operand 1 must be aligned on a 16-byte boundary. The format of operand 1 is as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length
0	0	Number of space location unlock requests in template	Bin(4)
4	4	Offset to lock state selection values	Bin(2)
6	6	Reserved (binary 0)	Char(26) +
32	20	Space location(s) to be unlocked (repeated as specified by *number of space location unlock requests in template* above)	[*] Space pointer
*	*	—- End —-	

The *unlock options* is located by adding the *offset to lock state selection values* above to operand 1.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Unlock options	[*] Char(1)	
		(repeated for each unlock request)		
0	0		Lock state to unlock	Bits 0-4
			(1 = unlock requested, 0 = unlock not requested) Only one state may be requested.	
0	0		LSRD lock	Bit 0
0	0		LSRO lock	Bit 1
0	0		LSUP lock	Bit 2
0	0		LEAR lock	Bit 3
0	0		LENR lock	Bit 4
0	0		Lock count option	Bit 5
			0 = Lock count reduced by 1	
			1 = All locks are unlocked. (The lock count is set to 0).	
0	0		Reserved (binary 0)	Bit 6
0	0		Entry active indicator	Bit 7
			0 = Entry not active. This entry is not used.	
			1 = Entry active. Lock is to be unlocked.	
*	*	—- End —-		

Note: Fields indicated with a plus sign (+) are ignored by the instruction.

This instruction can request the deallocation of one or more lock states on one or more space locations. The locks are deallocated sequentially until all specified locks are deallocated. When a lock is deallocated, the lock count is either reduced by 1 or set to 0 for the specified state. This option is specified by the **lock count option**.

Specific locks can be unlocked only if they are held by the thread issuing the Unlock Space Location instruction. If a space location lock cannot be unlocked because the thread does not hold the indicated lock, then exception data is saved but processing of the instruction continues. After all requested space location locks have been processed, the *invalid unlock request* (hex 1A03) exception is signaled if any space location lock was not unlocked.

No locks are unlocked if a template value is invalid.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

- 0601 Space Addressing Violation
- 0602 Boundary Alignment
- 0603 Range

08 Argument/Parameter

- 0801 Parameter Reference Violation

10 Damage Encountered

- 1004 System Object Damage State
- 1044 Partial System Object Damage

1A Lock State

- 1A03 Invalid Unlock Request
- 1A05 Invalid Space Location Unlocked

1C Machine-Dependent

- 1C03 Machine Storage Limit Exceeded
- 1C06 Machine Lock Limit Exceeded

20 Machine Support

- 2002 Machine Check
- 2003 Function Check

22 Object Access

- 2202 Object Destroyed
- 2208 Object Compressed
- 220B Object Not Available

24 Pointer Specification

- 2401 Pointer Does Not Exist
- 2402 Pointer Type Invalid

2E Resource Control Limit

- 2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

Unlock Teraspace Storage Location (UNLCKTSL)

Op Code (Hex)	Operand 1
03D5	Unlock request template

Operand 1: Space pointer data object.

Bound program access

```
Built-in number for UNLCKTSL is 622.
UNLCKTSL (
          unlock_request_template   : address of space pointer(16)
)
```

Description: The locks specified in the *unlock template* (operand 1) are unlocked. UNLCKTSL can unlock any location locked by the LOCKTSL or LOCKSL instruction. A maximum of 4093 locations can be unlocked with one UNLCKTSL instruction.

Any teraspace storage location(s) specified within the template must either be mapped or allocated, otherwise an *invalid unlock request* (hex 1A03) exception is signaled.

A location specified within the template which is not a teraspace storage location (i.e. is a single level store location) need not exist when this instruction is issued.

If a teraspace storage location lock cannot be unlocked because the thread does not hold the indicated lock, an *invalid unlock request* (hex 1A03) exception is signaled.

The lock request template identified by operand 1 must be aligned on a 16-byte boundary or an *boundary alignment* (hex 0602) exception is signaled.

The format of operand 1 is as follows:

Offset			
Dec	**Hex**	**Field Name**	**Data Type and Length**
0	0	Number of unlock requests in template	UBin(4)

Offset Dec	Offset Hex	Field Name	Data Type and Length	
4	4	Offset to lock state selection values	UBin(2)	
6	6	Reserved (binary 0)	Char(9)	
15	F	Lock request options	Char(1)	
15	F		Lock scope	Bit 0
			0 = Lock is scoped to the current thread	
			1 = Lock is scoped to the *lock scope object type*	
15	F		Lock scope object type	Bit 1
			0 = Process containing the current thread	
			1 = Transaction control structure attached to the current thread.	
15	F		Reserved (binary 0)	Bits 2-7
16	10	Reserved (binary 0)	Char(16)	
32	20	Location(s) to be unlocked	[*] Space pointer	
		(repeated as specified by *number of unlock requests in template* above)		
*	*	—- End —-		

The *unlock options* is located by adding the *offset to lock state selection values* above to operand 1.

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Unlock options	[*] Char(1)	
		(repeated for each unlock request)		
0	0		Lock state to unlock	Bits 0-4
			(1 = unlock requested, 0 = unlock not requested) Only one state may be requested.	
0	0		LSRD lock	Bit 0
0	0		LSRO lock	Bit 1
0	0		LSUP lock	Bit 2
0	0		LEAR lock	Bit 3
0	0		LENR lock	Bit 4
0	0		Lock count option	Bit 5
			0 = Lock count reduced by 1	
			1 = All locks are unlocked. (The lock count is set to 0).	
0	0		Reserved (binary 0)	Bit 6
0	0		Entry active indicator	Bit 7
			0 = Entry not active. This entry is not used.	
			1 = Entry active. Lock is to be unlocked.	
*	*	—- End —-		

This instruction can request the deallocation of one or more lock states on one or more locations. The locks are deallocated sequentially until all specified locks are deallocated. When a lock is deallocated, the lock count is either reduced by 1 or set to 0 for the specified state. This option is specified by the **lock count option**.

The **lock scope** field and the **lock scope object type** field determine which scope all specified unlock requests will be allocated to, either a thread, process or transaction control structure:

•

- When *lock scope* has a value of *lock is scoped to the lock scope object type* and *lock scope object type* has a value of *process containing the current thread*, the lock scope will be the process containing the current thread.
- When *lock scope* has a value of *lock is scoped to the lock scope object type* and *lock scope object type* has a value of *transaction control structure attached to the current thread*, the lock scope will be the transaction control structure that is attached to the current thread. If the current thread does not have a transaction control structure attached, then the lock scope will be the process containing the current thread.
- When *lock scope* has a value of *lock is scoped to the current thread*, the lock scope will be to the current thread.

If *lock scope object type* has a value of *transaction control structure attached to the current thread* and the transaction control structure state does not allow objects to be locked on behalf of the transaction control structure, a *object not eligible for operation* (hex 2204) exception is signaled.

Allocated process scoped locks and thread scoped locks, allocated by the initial thread of the process, are released when the process terminates. Allocated thread scoped locks are released when the thread terminates. If a thread requested a process scoped lock, the process will continue to hold that lock after termination of the requesting thread. If a thread requested a transaction control structure scoped lock, the transaction control structure will continue to hold that lock after the termination of the requesting thread.

Specific locks can be unlocked only if they are held by the thread issuing the Unlock Teraspace Storage Location instruction. If a space location lock cannot be unlocked because the thread does not hold the indicated lock, or if the address is a teraspace storage address which is not mapped or allocated, then exception data is saved but processing of the instruction continues. After all requested unlocks have been processed, an *invalid unlock request* (hex 1A03) exception is signaled if any location lock was not unlocked.

No locks are unlocked if a template value is invalid.

Warning: Temporary Level 3 Header

Authorization Required

•

- None

Lock Enforcement

•

- None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1A Lock State

1A03 Invalid Unlock Request

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

1C06 Machine Lock Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2204 Object Not Eligible for Operation

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

38 Template Specification

3801 Template Value Invalid

44 Protection Violation

Verify (VERIFY)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
VERIFY 10D7		Receiver	Source	Class	
VERIFYB 1CD7	Branch options	Receiver	Source	Class	Branch targets
VERIFYI 18D7	Indicator options	Receiver	Source	Class	Indicator targets

Operand 1: Binary variable scalar or binary array.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4-5:

-
- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: Each character of the *source* operand character string value is checked to verify that it is among the valid characters indicated in the *class* operand.

The operation begins at the left end of the *source* string and continues character by character, from left to right. Each character of the *source* value is compared with the characters of the *class* operand. If a match for the *source* character exists in the *class* string, the next *source* character is verified. If a match for the *source* character does not exist in the *class* string, the binary value for the relative location of the character within the *source* string is placed in the *receiver* operand.

If the *receiver* operand is a scalar, only the first occurrence of an invalid character is noted. If the *receiver* operand is an array, as many occurrences as there are elements in the array are noted.

The operation continues until no more occurrences of invalid characters can be noted or until the end of the *source* string is encountered. When the second condition occurs, the current *receiver* value is set to 0. If the *receiver* operand is an array, all its remaining entries are set to 0's.

The *source* and *class* operands can be variable length substring compound operands.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 2 and 3. The effect of specifying a null substring reference for the *class* operand when a nonnull string reference is specified for the *source* is that all of the characters of the *source* are considered invalid. In this case, the *receiver* is accordingly set with the offset(s) to the bytes of the *source*, and the instruction's resultant condition is positive. The effect of specifying a null substring reference for the *source* operand (no characters to verify) is that the *receiver* is set to zero and the instruction's resultant condition is zero regardless of what is specified for the *class* operand.

Resultant Conditions: The numeric value(s) of the *receiver* is either 0 or positive. When the *receiver* operand is an array, the resultant condition is 0 if all elements are 0.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1044 Partial System Object Damage

1C Machine-Dependent

1C03 Machine Storage Limit Exceeded

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2201 Object Not Found

2202 Object Destroyed

2203 Object Suspended

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2C Program Execution

2C04 Branch Target Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Wait On Time (WAITTIME)

Op Code (Hex)	Operand 1
0349	Wait template

Operand 1: Character(16) scalar.

Bound program access

```
Built-in number for WAITTIME is 66.
WAITTIME (
          wait_template   : address
)
```

Description: This instruction causes the thread to wait for a specified time interval. The current thread is placed in wait state for the amount of time specified by the *wait template* in accordance with the specified wait options.

The format of the *wait template* for operand 1 is:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Wait time interval	Char(8)	
8	8	Wait options	Char(2)	
8	8		Access state control for entering wait	Bit 0
			0 = Do not modify access state	
			1 = Modify access state	
8	8		Access state control for leaving wait	Bit 1
			0 = Do not modify access state	
			1 = Modify access state	
8	8		Multiprogramming level (MPL) control during wait	Bit 2

Offset Dec	Hex	Field Name	Data Type and Length	
			0 = Do not remain in current MPL set	
			1 = Remain in current MPL set	
8	8		Asynchronous signals processing option	Bit 3
			0 = Do not allow asynchronous signal processing during wait	
			1 = Allow asynchronous signal processing during wait	
8	8		Reserved (binary 0)	Bits 4-15
10	A	Reserved	Char(6)	
16	10	—- End —-		

See “Standard Time Format” on page 1272 for additional information on the format of the *wait time interval*.

The **access state control** options control whether the process access group (PAG) will be explicitly transferred between main and auxiliary storage when entering and leaving a wait as a result of execution of this instruction. Specification of *modify access state* requests that the PAG be purged from main to auxiliary storage for entering a wait and requests that the PAG be transferred from auxiliary to main storage for leaving a wait. Specification of *do not modify access state* requests that the PAG not be explicitly transferred between main and auxiliary storage as a result of executing this instruction.

The access state of the PAG is modified when entering the wait if the process is not multi-threaded (i.e., the waiting thread is the only thread in the process), if the process' instruction wait initiation access state control attribute specifies *allow access state modification*, if the *access state control for entering wait* option specifies *modify access state*, and if the *MPL control during wait* option specifies *do not remain in current MPL set*.

The **multiprogramming level (MPL) control during wait** option controls whether the thread will be removed from the current MPL set or remain in the current MPL set when the thread enters a wait as a result of executing this instruction.

When the *MPL control during wait option* specifies *remain in current MPL set* and the *access state control for entering wait* option specifies *do not modify access state*, the machine will check the wait time requested. If the wait time requested is less than an implementation-defined limit (which will not exceed 2 seconds), the thread will remain in the current MPL. If the wait time requested is greater than this limit, the *MPL control during wait option* is ignored and the thread is automatically removed from the MPL at the beginning of the wait. The automatic removal does not change or affect the total wait time specified for the thread in the wait time interval.

The **asynchronous signals processing option** controls the action to be taken if an asynchronous signal is pending or received while in wait. If an asynchronous signal that is not blocked or ignored is generated for the thread and the *asynchronous signals processing option* indicates *allow asynchronous signals processing during wait*, the wait will be terminated and an *asynchronous signal terminated MI wait* (hex 4C01) exception is signaled. Otherwise, when the *asynchronous signals processing option* indicates *do not allow asynchronous signals processing during wait*, the thread remains in the wait for the amount of time specified by the *wait time interval*.

While the thread is in wait state it may be interrupted for events unless the thread is masked.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

0603 Range

08 Argument/Parameter

0801 Parameter Reference Violation

10 Damage Encountered

1004 System Object Damage State

1005 Authority Verification Terminated Due to Damaged Object

1044 Partial System Object Damage

20 Machine Support

2002 Machine Check

2003 Function Check

22 Object Access

2202 Object Destroyed

2203 Object Suspended

2207 Authority Verification Terminated Due to Destroyed Object

2208 Object Compressed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

32 Scalar Specification

3201 Scalar Type Invalid

3202 Scalar Attributes Invalid

3203 Scalar Value Invalid

36 Space Management

3601 Space Extension/Truncation

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4C Signals Management

4C01 Asynchronous Signal Terminated MI Wait

X To The Y Power (POWER)

Bound program access

`Built-in number for POWER is 411.`
`POWER (`
`  source  : floating point(8) value`
`  exponent  : floating point(8) value`
`) :  floating point(8) value computed from source raised to the power`
`  exponent`

Description: The computation source ^{exponent} is performed and the result returned.

See floating point results from special values for additional information.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0C Computation

0C06 Floating-Point Overflow

0C07 Floating-Point Underflow

0C09 Floating-Point Invalid Operand

0C0D Floating-Point Inexact Result

0C0E Floating-Point Zero Divide

XOR (Exclusive Or) String (XORSTR)

Bound program access
Built-in number for XORSTR is 453. XORSTR (receiver_string : address of aggregate(*) first_source_string : address of aggregate(*) second_source_string : address of aggregate(*) string_length : unsigned binary(4,8) value which specifies the length in bytes of the three strings)

Description: Each byte value of the *first source string*, for the number of bytes indicated by *string length*, is logically **xor**ed (exclusive or) with the corresponding byte value of the *second source string*, on a bit-by-bit basis. The results are placed in the *receiver string*. If the strings overlap in storage, predictable results occur only if the overlap is fully coincident.

If the space(s) indicated by the three addresses are not long enough to contain the number of bytes indicated by *string length*, a *space addressing violation* (hex 0601) exception is signalled. Partial results in this case are unpredictable.

Warning: Temporary Level 3 Header

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

06 Addressing

0601 Space Addressing Violation

0602 Boundary Alignment

08 Argument/Parameter

0801 Parameter Reference Violation

22 Object Access

2202 Object Destroyed

220B Object Not Available

24 Pointer Specification

2401 Pointer Does Not Exist

2402 Pointer Type Invalid

44 Protection Violation

4401 Object Domain or Hardware Storage Protection Violation

4402 Literal Values Cannot Be Changed

Yield (YIELD)

Op Code (Hex)
0610

Bound program access

`Built-in number for YIELD is 539.`
`YIELD (`
`)`

Description: The dispatching algorithm is run. If another thread of equal or higher priority is eligible to run, then a thread is chosen and dispatched. Otherwise, the current thread resumes execution.

Warning: Temporary Level 3 Header

Usage Notes

The yield() function is a common technique used on other platforms to serialize on a resource or to allow other threads of equal or higher priority to execute before the current thread begins execution of a long running function.

•

- Serialization of a resource

  A ″spin″ lock is a high speed synchronization primitive in which the application ″loops″ on the setting of a variable which is used to synchronize access to a resource. A typical application implementation of a spin lock might be:

  1. Compare and swap on a variable that synchronizes access to a resource.
  2. If not available and first time in loop, invoke the yield() function.
  3. Otherwise if not the first time in loop, wait for a small time quantum. This time quantum is incremented each time through the loop.
  4. Loop back to the compare and swap statement.
- Allow threads of equal or higher priority to run

  The yield() function allows a thread to immediately relinquish control to a thread of equal or higher priority. On other platforms, this is done because the kernel can not (usually) be preempted. On

iSeries(TM), the duration of time another thread of equal or higher priority may be prevented from executing until the current thread reaches time-slice end is considered to be very large, especially considering the processing speeds of current machines.

Authorization Required

•

• None

Lock Enforcement

•

• None

Exceptions

•

• None

Concepts

These are the concepts for this category.

---

iSeries Machine Interface Introduction

Overview

This web page contains the following:

•

• Detailed descriptions of the iSeries machine interface instruction fields and the formats of these fields
• A description of the format used in describing each instruction
• A list of the terms in the syntax that define the characteristics of the operands
• A discussion of some pervasive topics that apply to a wide range of instructions.

You should read this web page in its entirety before attempting to write instructions.

What's New for V5R3

The following changes have been made to the MI architecture for V5R3:

1. 8 byte activation and invocation marks are now supported. If you are currently using 4 byte marks, it is suggested you change to use the 8 byte marks. 8 byte marks are supported on the following MI instructions:
   - "Activate Bound Program (ACTBPGM)" on page 5
   - "Find Relative Invocation Number (FNDRINVN)" on page 396
   - "Materialize Activation Attributes (MATACTAT)" on page 458
   - "Materialize Activation Export (MATACTEX)" on page 464
   - "Materialize Activation Group Attributes (MATAGPAT)" on page 466
   - "Materialize Activation Group-Based Heap Space Attributes (MATHSAT)" on page 472
   - "Materialize Invocation Attributes (MATINVAT)" on page 579
   - "Materialize Invocation Entry (MATINVE)" on page 591
   - "Materialize Invocation Stack (MATINVS)" on page 597
   - "Materialize Process Activation Groups (MATPRAGP)" on page 739
   - "Materialize Process Message (MATPRMSG)" on page 770
   - "Materialize Pointer (MATPTR)" on page 718
   - "Reinitialize Static Storage (RINZSTAT)" on page 1025
2. The "Compare and Swap (CMPSW)" on page 74 has a new operand that can be used to specify whether or not to perform storage synchronization.
3. The following instructions now support 63 digits for zoned decimal and packed decimal values:
   - "Copy Numeric Value (CPYNV)" on page 254
   - "Convert Character to Numeric (CVTCN)" on page 155
   - "Convert External Form to Numeric Value (CVTEFN)" on page 174
   - "Convert Numeric to Character (CVTNC)" on page 188
   - "Edit (EDIT)" on page 341
   - "Set Data Pointer Attributes (SETDPAT)" on page 1112
4. The "Lock Teraspace Storage Location (LOCKTSL)" on page 446 and "Unlock Teraspace Storage Location (UNLCKTSL)" on page 1250 instructions support locks scoped to a Transaction Control Structure.
5. The "Materialize Journal Port Attributes (MATJPAT)" on page 603 instruction supports a Quiesced Status field.
6. The maximum sequence number has been increased on the "Materialize Journal Space Attributes (MATJSAT)" on page 612.
7. The "Materialize Machine Data (MATMDATA)" on page 693 supports materializing the time-of-day clock as Coordinated Universal Time (UTC).
8. The "Materialize Process Mutex (MATPRMTX)" on page 788 can materialize additional mutex information.
9. The "Translate Multiple Bytes (XLATEMB)" on page 1217 supports UTF-8.
10. The "Materialize Resource Management Data (MATRMD)" on page 833 has new options to materialize the following information:
    - Dynamic thread resources affinity adjustment
    - ASP space information
    - Processor utilization data
    - Shared processor pool information
    - Multiprocessor utilizations
    - Machine resource portions
    - Interrupt polling control

11. The "Materialize Machine Attributes (MATMATR)" on page 619 has new options to materialize the following information:
    - Memory on demand
    - Hardware Management Console (HMC) Information
12. The following new instructions have been added:
    - "Atomic Add (ATMCADD)" on page 33
    - "Atomic And (ATMCAND)" on page 35
    - "Atomic Or (ATMCOR)" on page 37
    - "Call Program with Variable Length Argument List (CALLPGMV)" on page 48
    - "Check Lock Value (CHKLKVAL)" on page 50
    - "Clear Lock Value (CLRLKVAL)" on page 72
    - "Materialize Machine Information (MATMIF)" on page 697
    - "Materialize Time of Day Clock Attributes (MATTODAT)" on page 938

Instruction Format Conventions Used

The user must be aware that not every instruction uses every field described in this section. Only the information pertaining to the fields that are used by an instruction is provided for each instruction.

Each instruction is formatted with the instruction name followed by its base mnemonic. Following this, for instructions supported by Non-Bound Programs (NBP), is the operation code (op code) in hexadecimal and the number of operands with their general meaning.

Example:

ADD NUMERIC (ADDN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
1043	Sum	Addend 1	Addend 2

This information is followed by the operands and their syntax. See "Definition Of The NBP Operand Syntax" on page 1269 for a detailed discussion of the syntax of instruction operands.

Example:

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

For instructions that are supported in Bound Programs (BP), a bound program access box is provided that describes the operands and return values associated with the instruction in bound programs. For example:

Bound program access

```
Built-in number for ALCHSS is 111.
ALCHSS (
        heap_identifier            : signed binary(4) OR
                                     unsigned binary(4) OR
                                     null operand
        size_of_space_allocation   : signed binary(4)
) :   space pointer(16) to a space allocation
```

Note:	Within the bound program access box, a data type like signed binary(1,2,4) is a short hand notation for the 3 data types signed binary(1), signed binary(2), and signed binary(4) all being supported.

A description of the parameters for bound program access to the instruction is given. See the corresponding programming language reference manual for details as to how this information should be interpreted for a given language.

If an operand is passed by reference, the data type of the operand is preceded by the word address. If an operand is passed by value, the data type of the operand is not preceded by the word address. A local form address, which is a type of address that can only refer to teraspace, can be used on any instructions that specify an address operand unless explicitly prohibited.

An address is the value contained in any space pointer, the location of a data object or the result of an address computation.

If the built-in function has a return value, it is specified following the closing ') :' characters. Not all built-in functions have a return value. If the built-in function has a return value, it may be returned either by address or by value and will state which method is used.

Description: A detailed description and a functional definition of the instruction is given.

Note: When the description refers to a space pointer or procedure pointer and the length of the pointer is not explicitly given, the length should be inferred from the context. A pointer's length is explicitly specified in operand and template definitions.

When an instruction takes a template (data structure) as input or provides a template as output, the format of the template is defined as a series of fields. Each field is given a name and an associated data type. The valid data types are:

Bin(1)	Signed 1-byte binary (not valid if template is used by a non-bound program instruction).
UBin(1)	Unsigned 1-byte binary (not valid if template is used by a non-bound program instruction).
Bin(2)	Signed 2-byte binary
UBin(2)	Unsigned 2-byte binary
Bin(4)	Signed 4-byte binary
UBin(4)	Unsigned 4-byte binary
Bin(8)	Signed 8-byte binary (not valid if template is used by a non-bound program instruction).
UBin(8)	Unsigned 8-byte binary (not valid if template is used by a non-bound program instruction).
Bit x	A 1-bit field occupies position x. Bits are numbered from the highest most position being 0 (left-most) and the lowest most position being n (right-most).
Bit y-z	A series of 1-bit fields that occupy positions x through z inclusive. Bits are numbered from the highest most position being 0 (left-most) and the lowest most position being n (right-most).
Char(n)	Fixed length string of ″n″ 1-byte characters. Char(1) is also used to represent 1-byte binary values. Char(8) is used to represent 8-byte binary. The character string may be redefined to be series of 1 or more other data types.
Char(*)	Variable length string of 1-byte characters. The character string may be redefined to be series of 1 or more other data types.
Float(4)	A 4-byte floating point number.
Float(8)	An 8-byte floating point number.

Data pointer	A 16-byte area that contains a data pointer (must be on a 16-byte boundary).
Instruction pointer	A 16-byte area that contains an instruction pointer (must be on a 16-byte boundary).
Invocation pointer	A 16-byte area that contains an invocation pointer (must be on a 16-byte boundary).
Label pointer	A 16-byte area that contains a label pointer (must be on a 16-byte boundary).
Object pointer	A 16-byte area that contains an XOM object pointer (must be on a 16-byte boundary).
Procedure pointer(16)	A 16-byte area that contains a procedure pointer (must be on a 16-byte boundary).
Procedure pointer(8)	An 8-byte area that contains a local pointer that identifies an active instance of a procedure (not valid in a template for a non-bound program, and valid in a bound program template only when the program is to be created as teraspace capable)
Space pointer(16)	A 16-byte area that contains a space pointer (must be on a 16-byte boundary).
Space pointer(8)	An 8-byte area that contains a local pointer to teraspace (not valid in a template for a non-bound program, and valid in a bound program template only when the program is to be created as teraspace capable)
Suspend pointer	A 16-byte area that contains a system pointer (must be on a 16-byte boundary).
Synchronization pointer	A 16-byte area that contains a synchronization pointer (must be on a 16-byte boundary).
System pointer	A 16-byte area that contains a system pointer (must be on a 16-byte boundary).
Open pointer	A 16-byte area that contains a pointer with an unspecified type (must be on a 16-byte boundary).

Example:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Materialization size specification	Char(8)	
0	0		Number of bytes provided for materialization	UBin(4)
4	4		Number of bytes available for materialization	UBin(4)
8	8	Object identification	Char(32)	
8	8		Object type	Char(1)
9	9		Object subtype	Char(1)
10	A		Object name	Char(30)
40	28	Reserved (binary 0)	Char(4)	
44	2C	Performance class	Char(4)	
44	2C		Space alignment	Bit 0
44	2C		Reserved (binary 0)	Bits 1-4
44	2C		Main storage pool selection	Bit 5
44	2C		Transient storage pool selection	Bit 6
44	2C		Block transfer on implicit access state modification	Bit 7
44	2C		Unit number	Bits 8-15
44	2C		Reserved (binary 0)	Bits 16-31
48	30	Context	System pointer	
64	40	—- End —-		

This hypothetical template is composed of:

1. An 8-byte character string which is defined to be composed of 2 unsigned 4-byte binary values.

2. A 32-byte character string which is defined to be composed of three fields: a 1-byte character field, another 1-byte character field, and lastly a 30-byte character field.
3. A 4-byte character string which is defined to be reserved for future use and that will contain a value of binary zeroes.
4. A 4-byte character string which is defined to be composed of a series of bit fields: 1-bit fields are defined for positions 0, 5, 6, and 7; a 4-bit field is reserved in positions 1 through 4; an 8-bit field is defined in positions 8 through 15; and a 16-bit field is defined as being reserved in positions 16-31.
5. A system pointer to a context object. This pointer is on a 16-byte boundary with respect to the beginning of the template. The architecture assumes that the template begins on a 16-byte boundary.
6. The template is 64 bytes in length.

When a template field represents an array or repeating structure, the field data type will be proceeded by a ″dimension″ which indicates the number of elements in the array or the number of times the structure is repeated.

Example:

Offset				
Dec	**Hex**	**Field Name**	**Data Type and Length**	
0	0	Array of binary numbers	[12] Bin(4)	
48	30	Repeating structure	[4] Char(4)	
48	30		Structure field 1	Char(1)
49	31		Structure field 2	Char(1)
50	32		Structure field 3	Bin(2)
64	40	Array of pointers	[3] Space pointer	
112	70	Variable array 1	[5] Char(*)	
*	*	Variable array 2	[*] Char(*)	
*	*	—- End —-		

In this example, there is:

1. Twelve occurrences of a signed 4-byte binary number.
2. Four occurrences of a 4-byte structure which is composed of 3 fields; 2 1-byte character strings and 1 signed 2-byte binary number.
3. Three occurrences of a 16-byte space pointer.
4. Five occurrences of a variable length character string. Normally the instruction will provide additional information regarding how to determine the length of each occurrence.
5. A variable number of occurrences of a variable length character string. Normally the instruction will provide additional information regarding how to determine the number of occurrences and the length of each occurrence.

When terms are defined or fields in a template are described, they are highlighted as follows: **term definition**. When a term is referenced or a value of a field is referred to, it is highlighted as follows: *term reference*.

Fields in a template are generally described in the same order as they are defined in the template. However, some fields are more appropriately described with other related fields, so they may not appear in exact order.

Limitations (Subject to Change):

These are the limits that apply to the functions performed by the instruction. These limits are subject to change.

Resultant Conditions:

These are the conditions that can be set at the end of the standard operation in order to perform a conditional branch or set a conditional indicator.

Warning: Temporary Level 4 Header

Authorization Required: A list of the object authorization required for each of the operands in the instruction or for any objects subsequently referenced by the instruction is given.

Lock Enforcement: Describes the specification of the lock states that are to be enforced during execution of the instruction.

The following states of enforcement can be specified for an instruction:

-
- Enforcement for materialization

  Access to a system object is allowed if no other process is holding a locked exclusive no read (LENR) lock on the object. In general, this rule applies to instructions that access an object for materialization and retrieval.
- Enforcement for modification

  Access to a system object is allowed if no other process is holding a locked exclusive no read (LENR), locked exclusive allow read (LEAR), or locked shared read only (LSRO) lock. In general, this rule applies to instructions that modify or alter the contents of a system object.
- Enforcement of object control

  Access is prohibited if another process is holding any lock on the system object. In general, this rule applies to instructions that destroy or rename a system object.

Error Conditions: For those instructions supported in bound programs that return a completion value, this section lists the possible values that can be returned.

Exceptions: The ″exceptions″ sections contain a list of exceptions that can be caused by the instruction. Exceptions related to specific operands are indicated for each exception by the exception under the heading operand. An entry under the word, other, indicates that the exception applies to the instruction but not to a particular operand.

Reserved and Obsolete Fields

Reserved Fields

Fields in an instruction template that are specified as being reserved must contain binary 0s on input and will usually contain binary 0s on output. In most cases, specifying a value other than zero may cause an exception or unpredictable results.

As the MI Architecture evolves and responds to new and changing requirements, it is necessary to make use of reserved fields to support new function. The MI user must be aware that on subsequent releases

-
- a formerly reserved field may not return binary 0s
- a reserved field that was not checked for binary 0s may begin to have binary 0 values enforced on input
- a formerly reserved field may begin to accept nonzero values in support of a new or expanded function.

So, for example, if a template byte had the first six bits used, followed by two reserved bits, code that relies on having all the defined bits turned on should check the bits individually instead of comparing the whole byte to hex FC.

Obsolete Fields

As the MI Architecture evolves and responds to new and changing requirements, it is necessary sometimes to remove support for an existing field in a template. When this occurs, the field will remain in the template to provide compatibility with existing instances of the instruction but the field will be marked as obsolete.

Values provided by the MI user in obsolete fields will be ignored by the machine.

Definition Of The NBP Operand Syntax

Syntax consists of the allowable choices for each instruction operand. The following are the common terms used in the syntax and the meanings of those terms:

-
- *Numeric*: Numeric attribute of binary, packed decimal, zoned-decimal, or floating-point
- *Character*: character attribute
- *Scalar*:
  -
  - Scalar data object that is not an array (see note 1)
  - Constant scalar object
  - Immediate operand (signed or unsigned)
  - Element of an array of scalars (see notes 1 and 2)
  - Substring of a character scalar or a character scalar constant data object (see notes 1 and 3)
- *Data Pointer Defined Scalar*:
  -
  - A scalar defined by a data pointer
  - Substring of a character scalar defined by a data pointer (see notes 1 and 3)
- *Pointer*:
  -
  - Pointer data object that is not an array (see note 1)
  - Element of an array of pointers (see notes 1 and 2)
  - Space pointer machine object
- *Array*: An array of scalars or an array of pointers (see note 1)
- *Variable Scalar*: Same as scalar except constant scalar objects and immediate operand values are excluded.
- *Data Pointer*: A pointer data object that is to be used as a data pointer.
  -
  - If the operand is a source operand, the pointer storage form must contain a data pointer when the instruction is executed.
  - If the operand is a receiver operand, a data pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *Open pointer*: specifies that **all** 16-byte pointer types are supported by the instruction. When a new pointer type is added to the architecture, the instruction's implementation doesn't require any updates.

  However, if an instruction lists all of the defined 16-byte pointer types as valid data types for an operand, that instruction's implementation may have to be updated when a new pointer type is defined.
- *Space Pointer*: A space pointer data object or a space pointer machine object.
- *Space Pointer Data Object*: A pointer data object that is to be used as a space pointer.
  -

  - If the operand is a source operand, the pointer storage form must contain a space pointer when the instruction is executed.
  - If the operand is a receiver operand, a space pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *System Pointer*: a pointer data object that is to be used as a system pointer.
  -
  - If the operand is a source operand, the specified area must contain a system pointer when the instruction is executed.
  - If the operand is a receiver operand, a system pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *Relative Instruction Number*: Signed immediate operand. (NBP only)
- *Instruction Number*: Unsigned immediate operand. (NBP only)
- *Instruction Pointer*: A pointer data object that is to be used as an instruction pointer. (NBP only)
  -
  - If the operand is a source operand, the specified area must contain an instruction pointer when the instruction is executed.
  - If the operand is a receiver operand, an instruction pointer is constructed by the instruction in the specified area regardless of its current contents (see notes 4 and 5).
- *Invocation Pointer*: A pointer data object that is to be used as an invocation pointer.
  -
  - If the operand is a source operand, the specified area must contain an invocation pointer when the instruction is executed.
  - If the operand is a receiver operand, an invocation pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *Procedure Pointer*: A pointer data object that is to be used as a procedure pointer.
  -
  - If the operand is a source operand, the specified area must contain a procedure pointer when the instruction is executed.
  - If the operand is a receiver operand, a procedure pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *Label Pointer*: A pointer data object that is to be used as an label pointer.
  -
  - If the operand is a source operand, the specified area must contain a label pointer when the instruction is executed.
  - If the operand is a receiver operand, a label pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *Suspend Pointer*: A pointer data object that is to be used as an suspend pointer.
  -
  - If the operand is a source operand, the specified area must contain a suspend pointer when the instruction is executed.
  - If the operand is a receiver operand, a suspend pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *Synchronization Pointer*: A pointer data object that is to be used as an synchronization pointer.
  -
  - If the operand is a source operand, the specified area must contain a synchronization pointer when the instruction is executed.
  - If the operand is a receiver operand, a synchronization pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).

- *Object Pointer*: A pointer data object that is to be used as an XOM object pointer.
  –
  – If the operand is a source operand, the specified area must contain an XOM object pointer when the instruction is executed.
  – If the operand is a receiver operand, an XOM object pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *Instruction Definition List Element*: An entry in an instruction definition list that can be used as a branch target (NBP only). A compound subscript operand form must always be used (see note 5).

Notes:

1. An instruction operand in which the primary operand is a scalar or a pointer may also have an operand form in which an explicit base pointer is specified.

   See ODT Object References for more information on compound operands.
2. A compound subscript operand may be used to select a specific element from an array of scalars or from an array of pointers.

   See ODT Object References for more information on compound operands.
3. A compound substring operand may be used to define a substring of a character scalar, or a character constant scalar object.

   A compound substring operand that disallows a null substring reference (a length value of zero) may, unless precluded by the particular instruction, be specified for any operand syntactically defined as allowing a character scalar. A compound substring operand that allows a null substring reference may be specified for an operand syntactically defined as allowing a character scalar only if the instruction specifies that it is allowed. Whether a compound substring operand does or does not allow a null substring reference is controlled through the specification of the length secondary operand field.

   See ODT Object References for more information on compound operands.
4. A compound subscript operand form may be used to select an element from an array of pointers to act as the operand for an instruction. See ODT Object References for more information on compound operands.
5. Compound subscript forms are not allowed on branch target operands that are used for conditional branching. Selection of elements of instruction pointer arrays and elements of instruction definition lists may, however, be referenced for branch operands by the branch instruction.

Alternate choices of NBP operand types and the allowable variations within each choice are indicated in the syntax descriptions as shown in the following example.

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Instruction number, branch point or instruction pointer.

Operand 1 must be variable scalar. Operands 1 and 2 must be numeric. Operand 3 can be an instruction number, branch point or instruction pointer.

When a length is specified in the syntax for an NBP operand, character scalar operands must be at least the size specified. Any excess beyond that required by the instruction is ignored.

Scalar NBP operands that are operated on by instructions requiring 1-byte operands, such as pad values or indicator operands, can be greater than 1 byte in length; however, only the first byte of the character string is used. The remaining bytes are ignored by the instruction.

Names

The MI architecture treats names, unless explicitly stated otherwise, as a sequence of unencoded bytes. That is, the machine treats each byte of an object name as a sequence of bits which the machine stores and returns without modification.

Names, being unencoded, are not associated with any Coded Character Set Identifier (CCSID) or any other National Language Support information by the machine. The machine does not perform (unless explicitly stated) any character translations on names.

Character Constants

The character constants specified in an instruction (in the form 'X' where X represents a single character) are required by the machine to be from the EBCDIC invariant character set. This includes the following characters.

```
'A' = hex C1     'a' = hex 81     '0' = hex F0      ':' = hex 7A
'B' = hex C2     'b' = hex 82     '1' = hex F1      ';' = hex 5E
'C' = hex C3     'c' = hex 83     '2' = hex F2      '?' = hex 6F
'D' = hex C4     'd' = hex 84     '3' = hex F3
'E' = hex C5     'e' = hex 85     '4' = hex F4
'F' = hex C6     'f' = hex 86     '5' = hex F5
'G' = hex C7     'g' = hex 87     '6' = hex F6
'H' = hex C8     'h' = hex 88     '7' = hex F7
'I' = hex C9     'i' = hex 89     '8' = hex F8
'J' = hex D1     'j' = hex 91     '9' = hex F9
'K' = hex D2     'k' = hex 92     '+' = hex 4E
'L' = hex D3     'l' = hex 93     '>' = hex 6E
'M' = hex D4     'm' = hex 94     '=' = hex 7E
'N' = hex D5     'n' = hex 95     '<' = hex 4C
'O' = hex D6     'o' = hex 96     '%' = hex 6C
'P' = hex D7     'p' = hex 97     '@' = hex 7C
'Q' = hex D8     'q' = hex 98     '*' = hex 5C
'R' = hex D9     'r' = hex 99     ''' = hex 7D
'S' = hex E2     's' = hex A2     '"' = hex 7F
'T' = hex E3     't' = hex A3     '(' = hex 4D
'U' = hex E4     'u' = hex A4     ')' = hex 5D
'V' = hex E5     'v' = hex A5     ',' = hex 6B
'W' = hex E6     'w' = hex A6     '_' = hex 6D
'X' = hex E7     'x' = hex A7     '-' = hex 60
'Y' = hex E8     'y' = hex A8     '.' = hex 4B
'Z' = hex E9     'z' = hex A9     '/' = hex 61
```

Standard Time Format

The **Standard Time Format** is defined as a 64-bit (8-byte) unsigned binary value as follows:

Offset Dec	Offset Hex	Field Name	Data Type and Length	
0	0	Standard Time Format	UBin(8)	
0	0		Time	Bits 0-48
0	0		Uniqueness bits	Bits 49-63
8	8	—- End —-		

The **time** field is a binary number which can be interpreted as a time value in units of 8 microseconds. A binary 1 in bit 48 is equal to 8 microseconds.

The **uniqueness bits** field may contain any combination of binary 1s and 0s. These bits do not provide additional granularity for a time value; they merely allow unique 64-bit values to be returned, such as when the value of the *time-of-day (TOD) clock* is materialized.

A number of MI instructions define fields to contain a binary value which may represent a time stamp or time interval, or may specify a wait time-out period. Unless explicitly stated otherwise, the format of the field is the *Standard Time Format*.

Examples of binary values as time intervals:

```
A hex value of...  Represents...
0000000000008000   8 microseconds
00000000F4240000   1 second
00000D693A400000   1 hour
0008CD0E3A000000   1 week
```

Examples of binary values as time stamps:

```
A hex value of...  Represents...
0000000000000000   08/23/1928 12:03:06.314752
4A2FEC4C82000000   01/01/1970 00:00:00.000000
8000000000000000   01/01/2000 00:00:00.000000
DFFFFFFFFFF8000    07/07/2053 20:57:40.263928
```

Time-of-Day (TOD) Clock

The **time-of-day (TOD) clock** is a machine facility which provides a consistent measure of elapsed time. The value of the *TOD clock* can be materialized in "Standard Time Format" on page 1272. The *time* field of the *TOD clock* is incremented by adding a binary 1 in bit 48 every 8 microseconds. This gives the *TOD clock* a granularity of 8 microseconds. However, the observed granularity cannot be accurately predicted because retrieval latency depends on the current implementation of the machine facility and the workload on the machine when the request is made.

Depending upon the MI instruction used, the value of the *TOD clock* may be materialized as either the *Coordinated Universal Time (UTC)* for the system, with or without a *time zone offset*, or the *local time* for the system.

Coordinated Universal Time (UTC) is a universally coordinated time scale that is kept by timing laboratories around the world and is determined using highly precise atomic clocks. The International Bureau of Weights and Measures makes use of data from the timing laboratories to provide the international standard UTC. UTC is the current term for what was commonly referred to as Greenwich Mean Time (GMT). Zero hours UTC is midnight in Greenwich, England, which lies on the zero longitudinal meridian.

The **time zone offset** is a signed integer which indicates the local time zone, including any adjustment for Daylight Savings Time, as measured in minutes of time westward from Greenwich, England. The *time zone offset* can be used to convert between *UTC* and *local time* for the system.

The **local time** for the system is generated by applying the *time zone offset* for the system to the *UTC* for the system. When the value of the *time zone offset* is binary 0, then the *UTC* and the *local time* for a system are identical.

Both the value of the *TOD clock* as *UTC* and the *time zone offset* for the system can be materialized using the Materialize Time Of Day Attributes (MATTODAT) instruction.

The value of the *TOD clock* as either *UTC* or as *local time* for the system can be materialized using the Materialize Machine Data (MATMDATA) instruction.

The *TOD clock* machine facility guarantees that each request to materialize the value of the *TOD clock* will receive a unique, monotonically increasing value. However, the MI user must be aware that under the following conditions unique, monotonically increasing values may not be observed:

-
- Values across threads

Each request may have a different latency between when the request is made and when the value is returned. Therefore, the values returned for requests from different threads may not be in the same order in which the requests are made.

- Modification of the *TOD clock*

  Modification of the *TOD clock* to an earlier value can result in the materialization of values which are not unique nor monotonically increasing as compared to values materialized before the modification.

- *Time zone offset* change

  A *time zone offset* change, such as when changing from Daylight Saving Time to Standard Time, can result in the materialization of *local time* values which are not unique nor monotonically increasing as compared to *local time* values materialized before the change.

Storage Terminology

The term **basic storage unit** is defined to mean 512 bytes of storage. Basic storage units are commonly used to return the size information about MI objects.

The term **page** is defined to mean one or more *basic storage units* used by the machine to manage memory and DASD. The number of bytes in a *page* can be determined with option hex 12 of the Materialize Resource Machine Data (MATRMD) instruction.

The term **machine minimum transfer size** is defined to be the smallest number of bytes that may be transferred as a block to and from main storage. The number of bytes in the *machine minimum transfer size* can be determined with option hex 9 of the Materialize Resource Machine Data (MATRMD) instruction.

Storage Limitations

The following sub-sections describe data object size limits that are checked during module or program creation and storage limits that are checked when the program is activated or run. In some cases the creation-time limits are more generous than the run-time limits, so it's possible to create a program that will not run. The values in the following tables are in bytes.

Throughout this section, the following abbreviations are used:

K = 1024, M = 1048576, G = 1073741824, page = page size.

See "Storage Terminology" for more details on the page size.

Size limits for data objects in bound programs

Storage type	Program attribute		
	Not teraspace capable	Teraspace capable	Teraspace storage model
Automatic	16M-1page	16M-1page	2G-64K-1
Procedure argument block	16M-1page	16M-1page	2G-64K-1
Static	16M-1page	16M-1page	4G-1
Exported	16M-1page	16M-1page	16M-1page
Imported	16M-1page	16M-1page	16M-1page
Literal (constant)	16M-1page	16M-1page	16M-1page
Not mapped (based)	16M-1K-1	4G-1	4G-1

Size limits for data objects in non-bound programs

Storage type	Program attribute	
	Not teraspace capable	Teraspace capable
Automatic	16M-1page	16M-1page
Static (internal)	16M-1page	16M-1page
Static (named external)	64K-1	64K-1
Constant	32K-1	32K-1
Based (space pointer)	16M-1K-1	16M-1K-1
Based (PCO(Process Communications Object))	16M-1page	16M-1page
Parameter	16M-1K-1 *	16M-1K-1 *
Note: * Assumes parameter allocated in program-managed storage.		

Machine managed storage limits

Storage type	Program storage model	
	Single Level Store	Teraspace
Automatic stack, initial thread	16M-1page	64M
Automatic stack, secondary thread	16M-1page	16M
Static per compilation unit (bound)	16M-1page-16	4G-1
Static in activation group	2G-1	approx. 512G
Static constants per non-bound program	16M-1page	16M-1page
Static constants per bound program	approx. 256G	approx. 256G

Program managed storage limits

Storage type	Maximum # bytes
Pointer-based heap allocation	16M-1p (see ALCHSS)
Pointer-based heap	4G-512K (see CRTHS)
Space object, associated space	16M-1 page *
PCO (Process Communications Object)	16M-1 page
Note: * This is the maximum recommended size, for best access performance. The absolute maximum for a space object is 16M-256 and for an associated space of another type of object, 16M-32, depending upon the alignment chosen when the space or associated space is created.	

Atomicity

Atomicity of MI Instructions

MI instructions are **not** atomic unless they explicitly state that they perform some function atomically. So, it is possible for the function performed by an MI instruction to be only partially completed from the viewpoint of a program in another thread, or within the same thread when an exception occurs.

When a program is being created, many hardware instructions may be generated that will perform one MI instruction's function at run time. Without the use of an external control mechanism such as a locking protocol, sequences of instructions cannot be atomic. Further, the set of hardware instructions that implement an MI instruction, generated when a module or program is created, can be changed over time to provide more efficient programs or to accommodate new hardware. Thus even those MI instructions for which the generated code appears to be atomic should not be assumed to be so unless they are specified as performing a function atomically.

Atomicity of Storage Operations

A storage operation reads data from storage or writes data to storage. An MI instruction's function may use from zero to very many storage operations. An individual storage operation may be atomic or not. An atomic storage operation is performed such that it appears to be either complete or not yet started to all possible observers (i.e. all threads, whether on the same or another processor). On the other hand, storage operations that are not atomic may appear to be partially performed to some observers.

Atomicity can affect the use of shared storage areas. If a program modifies shared data while holding a lock that guarantees exclusive access (only one reader or writer can concurrently access the data) then atomicity is not an issue. However, if programs do not have exclusive access, then the atomicity of storage operations may be important. For example, a program running in one thread may periodically update a value in a space, which is in turn read by a program running in another thread. If the operation used to store a new value is atomic, then the other threads will always observe either the old value or the new value. If the operation used to store a new value is not atomic, then the other threads may observe the old value, the new value, or some composite consisting of part of the old value and part of the new value.

The MI makes only these guarantees regarding the atomicity of storage operations.

-
- a storage operation on a 16 byte MI pointer is atomic
- a storage operation that directly[1] (page 1277) refers to a local pointer aligned on an 8 byte boundary is atomic
- a storage operation that directly[1] (page 1277) refers to a binary data object aligned on a boundary that is a multiple of its length is atomic
- the storage update performed by a Compare and Swap instruction is atomic
- the storage update performed by the Atomic Add instruction is atomic
- the storage update performed by the Atomic And instruction is atomic
- the storage update performed by the Atomic Or instruction is atomic
- the storage update performed by the Check Lock Value instruction is atomic
- the storage update performed by the Clear Lock Value instruction is atomic

Note that operations on character data are guaranteed to be atomic only when Compare and Swap is used. Also note that, even though individual storage operations on properly aligned binary data objects are atomic, MI instructions operating on such data are not atomic (unless their descriptions explicitly say that they are). For example, a CPYNV from one 4 byte binary data object to another, where both are aligned on 4 byte boundaries, is **not** guaranteed to be an atomic operation, even though the read of one data object and write of the other are each individually atomic.

Shared Storage Access Ordering

Previous sections discussed atomicity of MI instructions and individual storage operations. For a related discussion of the ordering of multiple operations that access[2] (page 1277) **shared**[3] (page 1277) storage, see Storage Synchronization Concepts

External Standards and Architectures

Some of the instructions may make reference to external standards and architectures. To fully understand the functions performed by those instructions, it may be necessary to obtain a copy of the pertinent document.

Logical partitioning

Support for logical partitions has evolved since its initial release. Most MI instructions work the same in all logical partitions. However, a few MI instructions work differently depending on the logical

partitioning environment. Any differences are described in the documentation for the MI instruction. MATMATR option 01E0 can be used to materialize the firmware level and other logical paritioning information.

The following list describes some of the terms used with logical partitioning:

Firmware level	The level of the Licensed Internal Code used by the hypervisor.
Full System Partition Mode	A physical machine with a single partition owning all system resources.
Hardware Management Console	Attached appliance designed to control the physical machine. It is used to define and manage logical partitions and other system wide functions. If the physical machine has Hardware Management Console, some system attributes must be set here. Hardware Management Console is not supported on all firmware levels.
Hypervisor	A hypervisor is a specialized portion of the machine that enables logical partitioning.
Logical partition	A subset of a single physical machine that contains resources (processors, memory, input/output devices). A logical partition operates as an independent system. If hardware requirements are met, multiple logical partitions can exist within a physical machine.
Logical Partition Mode	A physical machine with firmware level hex 10 and one or more partitions owning system resources. Hardware Management Console is used to enter and exit this mode.
Primary logical partition	A logical partition which provides certain general functions on which all logical partitions are dependent. A primary logical partition is not supported on all firmware levels.
Secondary logical partition	A logical partition which maintains a dependency on the primary logical partition. Otherwise, it operates as a stand-alone system. Secondary logical partitions are independent of each other. Secondary logical partitions are not supported on all firmware levels.

The following list describes the logical partitioning environments:

Physical machine with firmware level hex 00	A primary logical partition exists. Some system attributes must be set in the primary logical partition only. The physical machine may have zero or more secondary logical partitions.
Physical machine with firmware level hex 10, no Hardware Management Console	A primary partition does NOT exist. System attributes must be set in a partition. This environment is in Full System Partition Mode.
Physical machine with firmware level hex 10, Hardware Management Console attached	A primary partition does NOT exist. Some system attributes must be set in a partition and other system attributes must be set using Hardware Management Console. This environment may be in Full System Partition Mode or Logical Partition Mode.

Footnotes:

[1] Direct references do not include operations that reference the storage containing a data object by using another view, such as a reference to a structure that contains the data object or a reference to the address of a storage range that contains the data object.

[2] (load from or store to)

[3] (used by more than one thread)

iSeries(TM) Machine Interface Instructions Sorted by Topic

Within each topic, the instructions are listed in alphabetical order.

Introduction

-
- "iSeries Machine Interface Introduction" on page 1262

Computation and Branching

-
- Arithmetic Operations
- Array Index Operations
- Boolean Operations
- “Add Logical Character (ADDLC)” on page 13
- “Add Numeric (ADDN)” on page 15
- “And (AND)” on page 24
- “Branch (B)” on page 39
- “Cipher (CIPHER)” on page 53
- “Clear Bit in String (CLRBTS)” on page 68
- “Compare Bytes Left-Adjusted (CMPBLA)” on page 81
- “Compare Bytes Left-Adjusted with Pad (CMPBLAP)” on page 83
- “Compare Bytes Right-Adjusted (CMPBRA)” on page 85
- “Compare Bytes Right-Adjusted with Pad (CMPBRAP)” on page 88
- “Compare Numeric Value (CMPNV)” on page 91
- “Compare and Swap (CMPSW)” on page 74
- “Compress Data (CPRDATA)” on page 110
- “Compute Array Index (CAI)” on page 113
- “Compute Math Function Using One Input Value (CMF1)” on page 119
- “Compute Math Function Using Two Input Values (CMF2)” on page 126
- “Concatenate (CAT)” on page 137
- “Convert BSC to Character (CVTBC)” on page 139
- “Convert Character to BSC (CVTCB)” on page 143
- “Convert Character to Hex (CVTCH)” on page 147
- “Convert Character to MRJE (CVTCM)” on page 149
- “Convert Character to Numeric (CVTCN)” on page 155
- “Convert Character to SNA (CVTCS)” on page 158
- “Convert Decimal Form to Floating-Point (CVTDFFP)” on page 172
- “Convert External Form to Numeric Value (CVTEFN)” on page 174
- “Convert Floating-Point to Decimal Form (CVTFPDF)” on page 178
- “Convert Hex to Character (CVTHC)” on page 182
- “Convert MRJE to Character (CVTMC)” on page 183
- “Convert Numeric to Character (CVTNC)” on page 188
- “Convert SNA to Character (CVTSC)” on page 191
- “Copy Bits Arithmetic (CPYBTA)” on page 208
- “Copy Bits Logical (CPYBTL)” on page 210
- “Copy Bits with Left Logical Shift (CPYBTLLS)” on page 212
- “Copy Bits with Right Arithmetic Shift (CPYBTRAS)” on page 214
- “Copy Bits with Right Logical Shift (CPYBTRLS)” on page 217
- “Copy Bytes Left-Adjusted (CPYBLA)” on page 220
- “Copy Bytes Left-Adjusted with Pad (CPYBLAP)” on page 222
- “Copy Bytes Overlap Left-Adjusted (CPYBOLA)” on page 224
- “Copy Bytes Overlap Left-Adjusted with Pad (CPYBOLAP)” on page 226
- “Copy Bytes Repeatedly (CPYBREP)” on page 229

Bound Program Computation and Branching Built-in Functions

-
- Floating point results from special values
- “And Complemented String (ANDCSTR)” on page 27
- “AND String (ANDSTR)” on page 28
- “Arc Cosine (ACOS)” on page 29
- “Arc Sine (ASIN)” on page 30
- “Arc Tangent (ATAN)” on page 31
- “Arc Tangent Hyperbolic (ATANH)” on page 32
- “Compute Length of Null-Terminated String (STRLENNULL)” on page 118
- Compare Null-Terminated Strings (STRCMPNULL)
- “Compare Null-Terminated Strings Constrained (STRNCMPNULL)” on page 90
- “Compare To Pad (CMPTOPAD)” on page 108
- “Complement String (COMSTR)” on page 109
- “Cosine (COS)” on page 264
- “Cosine Hyperbolic (COSH)” on page 265
- “Cotangent (COT)” on page 266
- “Copy Bytes (CPYBYTES)” on page 219
- “Copy Bytes Overlapping (CPYBO)” on page 228
- Copy Null-Terminated String (STRCPYNULL)
- “Copy Null-Terminated String Constrained (STRNCPYNULL)” on page 252
- “Copy Null-Terminated String Constrained, Null Padded (STRNCPYNULLPAD)” on page 253
- “Exponential Function of E (EEXP)” on page 379
- “Find Byte (FINDBYTE)” on page 390
- “Find Character Constrained (MEMCHR)” on page 391
- Find Character in Null-Terminated String (STRCHRNULL)
- “Logarithm Base E (Natural Logarithm) (LN)” on page 453
- “Memory Compare (MEMCMP)” on page 954
- “Memory Copy (MEMCPY)” on page 955
- “Memory Move (MEMMOVE)” on page 956
- “OR String (ORSTR)” on page 1019
- “Propagate Byte (PROPB)” on page 1022
- “Retrieve Computational Attributes (RETCA)” on page 1049
- “Scan Extended (SCANX)” on page 1072
- “Set Computational Attributes (SETCA)” on page 1107
- “Sine (SIN)” on page 1138
- “Sine Hyperbolic (SINH)” on page 1139
- “Tangent (TAN)” on page 1160
- “Tangent Hyperbolic (TANH)” on page 1161
- “Test and Replace Bytes (TESTRPL)” on page 1162
- “Test Subset (TESTSUBSET)” on page 1191
- “Translate Bytes (XLATEB)” on page 1214
- “Translate Bytes One Byte at a Time (XLATEB1)” on page 1215
- “XOR (Exclusive Or) String (XORSTR)” on page 1260
- “X To The Y Power (POWER)” on page 1259

Date/Time/Timestamp

-
- Date/Time Concepts
- “Compute Date Duration (CDD)” on page 115
- “Compute Time Duration (CTD)” on page 131
- “Compute Timestamp Duration (CTSD)” on page 134
- “Convert Date (CVTD)” on page 168
- “Convert Time (CVTT)” on page 202
- “Convert Timestamp (CVTTS)” on page 205
- “Decrement Date (DECD)” on page 300
- “Decrement Time (DECT)” on page 304
- “Decrement Timestamp (DECTS)” on page 307
- “Increment Date (INCD)” on page 408
- “Increment Time (INCT)” on page 412
- “Increment Timestamp (INCTS)” on page 415
- “Materialize Time of Day Clock Attributes (MATTODAT)” on page 938

Pointer/name resolution

-
- “Compare Pointer for Object Addressability (CMPPTRA)” on page 94
- “Compare Pointer for Space Addressability (CMPPSPAD)” on page 97
- “Compare Pointers for Equality (CMPPTRE)” on page 103
- “Compare Pointer Type (CMPPTRT)” on page 100
- “Copy Bytes with Pointers (CPYBWP)” on page 238
- “Materialize Pointer (MATPTR)” on page 718
- “Materialize Pointer Information (MATPTRIF)” on page 729
- “Materialize Pointer Locations (MATPTRL)” on page 736
- “Resolve Data Pointer (RSLVDP)” on page 1035
- “Resolve System Pointer (RSLVSP)” on page 1038
- “Retrieve Teraspace Address From Space Pointer (RETTSADR)” on page 1054
- “Set Object Pointer from Pointer (SETOBPFP)” on page 1120
- “Set Space Pointer from Pointer (SETSPPFP)” on page 1124
- “Set System Pointer from Pointer (SETSPFP)” on page 1131
- “Test Pointer (TESTPTR)” on page 1190

Space Addressing

-
- “Add Space Pointer (ADDSPP)” on page 19
- “Compare Space Addressability (CMPSPAD)” on page 106
- “Set Data Pointer (SETDP)” on page 1108
- “Set Data Pointer Addressability (SETDPADR)” on page 1110
- “Set Data Pointer Attributes (SETDPAT)” on page 1112
- “Set Space Pointer (SETSPP)” on page 1122
- “Set Space Pointer Offset (SETSPPO)” on page 1126
- “Set Space Pointer with Displacement (SETSPPD)” on page 1129

- “Store Space Pointer Offset (STSPPO)” on page 1146
- “Subtract Space Pointer Offset (SUBSPP)” on page 1155
- “Subtract Space Pointers For Offset (SUBSPPFO)” on page 1157

Space Management

-
- “Create Space (CRTS)” on page 285
- “Destroy Space (DESS)” on page 331
- “Materialize Space Attributes (MATS)” on page 916
- “Modify Space Attributes (MODS)” on page 971

Heap Management

-
- “Allocate Activation Group-Based Heap Space Storage (ALCHSS)” on page 21
- “Create Activation Group-Based Heap Space (CRTHS)” on page 266
- “Destroy Activation Group-Based Heap Space (DESHS)” on page 324
- “Free Activation Group-Based Heap Space Storage (FREHSS)” on page 403
- “Free Activation Group-Based Heap Space Storage From Mark (FREHSSMK)” on page 404
- “Materialize Activation Group-Based Heap Space Attributes (MATHSAT)” on page 472
- “Reallocate Activation Group-Based Heap Space Storage (REALCHSS)” on page 1023
- “Set Activation Group-Based Heap Space Storage Mark (SETHSSMK)” on page 1101

Program Management

-
- “Materialize Bound Program (MATBPGM)” on page 507
- “Materialize or Verify Licensed Internal Code Options (MVLICOPT)” on page 713
- “Materialize Program (MATPG)” on page 800
- “Materialize Program Name (MATPGMNM)” on page 820

Program Execution

-
- “Activate Bound Program (ACTBPGM)” on page 5
- “Activate Program (ACTPG)” on page 10
- “Call External (CALLX)” on page 41
- “Call Internal (CALLI)” on page 46
- “Call Program with Variable Length Argument List (CALLPGMV)” on page 48
- “Clear Invocation Exit (CLRIEXIT)” on page 70
- “Clear Invocation Flags (CLRINVF)” on page 71
- “Deactivate Program (DEACTPG)” on page 295
- “End (END)” on page 368
- “Find Relative Invocation Number (FNDRINVN)” on page 396
- “Invocation Pointer (INVP)” on page 423
- “Materialize Activation Attributes (MATACTAT)” on page 458
- “Materialize Activation Export (MATACTEX)” on page 464
- “Materialize Activation Group Attributes (MATAGPAT)” on page 466

- “Materialize Invocation (MATINV)” on page 574
- “Materialize Invocation Attributes (MATINVAT)” on page 579
- “Materialize Invocation Entry (MATINVE)” on page 591
- “Materialize Invocation Stack (MATINVS)” on page 597
- “Modify Automatic Storage Allocation (MODASA)” on page 957
- “NPM Procedure Parameter List Address (NPM_PARMLIST_ADDR)” on page 1013
- “OPM Parameter Address (OPM_PARM_ADDR)” on page 1015
- “OPM Parameter Count (OPM_PARM_CNT)” on page 1015
- “Reinitialize Static Storage (RINZSTAT)” on page 1025
- “Retrieve Invocation Flags (RETINVF)” on page 1054
- “Return External (RTX)” on page 1058
- “Set Argument List Length (SETALLEN)” on page 1103
- “Set Invocation Exit (SETIEXIT)” on page 1117
- “Set Invocation Flags (SETINVF)” on page 1120
- “Store Parameter List Length (STPLLEN)” on page 1144
- “Transfer Control (XCTL)” on page 1201

Program creation control

-
- “No Operation (NOOP)” on page 1010
- “No Operation and Skip (NOOPS)” on page 1010
- “Override Program Attributes (OVRPGATR)” on page 1020

Independent Index

-
- “Create Independent Index (CRTINX)” on page 271
- “Destroy Independent Index (DESINX)” on page 326
- “Find Independent Index Entry (FNDINXEN)” on page 391
- “Insert Independent Index Entry (INSINXEN)” on page 419
- “Materialize Independent Index Attributes (MATINXAT)” on page 560
- “Modify Independent Index (MODINX)” on page 965
- “Remove Independent Index Entry (RMVINXEN)” on page 1032

Queue Management

-
- “Dequeue (DEQ)” on page 311
- “Enqueue (ENQ)” on page 369
- “Materialize Queue Attributes (MATQAT)” on page 822
- “Materialize Queue Messages (MATQMSG)” on page 829

Object Lock Management

-
- “Lock Object (LOCK)” on page 424
- “Lock Object Location (LOCKOL)” on page 431
- “Lock Space Location (LOCKSL)” on page 440

- “Lock Teraspace Storage Location (LOCKTSL)” on page 446
- “Materialize Allocated Object Locks (MATAOL)” on page 477
- “Materialize Data Space Record Locks (MATDRECL)” on page 547
- “Materialize Object Locks (MATOBJLK)” on page 708
- “Materialize Process Locks (MATPRLK)” on page 767
- “Materialize Process Record Locks (MATPRECL)” on page 795
- “Materialize Selected Locks (MATSELLK)” on page 912
- “Transfer Object Lock (XFRLOCK)” on page 1206
- “Unlock Object (UNLOCK)” on page 1237
- “Unlock Object Location (UNLOCKOL)” on page 1241
- “Unlock Space Location (UNLOCKSL)” on page 1246
- “Unlock Teraspace Storage Location (UNLCKTSL)” on page 1250

Mutex Management

-
- “Create Pointer-Based Mutex (CRTMTX)” on page 281
- “Destroy Pointer-Based Mutex (DESMTX)” on page 328
- “Lock Pointer-Based Mutex (LOCKMTX)” on page 435
- “Materialize Mutex (MATMTX)” on page 704
- “Materialize Process Mutex (MATPRMTX)” on page 788
- “Unlock Pointer-Based Mutex (UNLKMTX)” on page 1244

Shared Storage Synchronization

-
- “Atomic Add (ATMCADD)” on page 33
- “Atomic And (ATMCAND)” on page 35
- “Atomic Or (ATMCOR)” on page 37
- “Check Lock Value (CHKLKVAL)” on page 50
- “Clear Lock Value (CLRLKVAL)” on page 72
- Shared Storage Synchronization Concepts
- “Synchronize Shared Storage Accesses (SYNCSTG)” on page 1159

Exception Management

-
- “Initialize Exception Handler Control Actions (INITEHCA)” on page 419
- “Materialize Exception Description (MATEXCPD)” on page 556
- “Modify Exception Description (MODEXCPD)” on page 962
- “Retrieve Exception Data (RETEXCPD)” on page 1050
- “Return From Exception (RTNEXCP)” on page 1060
- “Sense Exception Description (SNSEXCPD)” on page 1090
- “Signal Exception (SIGEXCP)” on page 1133
- “Test Exception (TESTEXCP)” on page 1175

Queue Space Management

-
- "Materialize Process Message (MATPRMSG)" on page 770

Context Management

-
- "Materialize Context (MATCTX)" on page 539

Authorization Management

-
- "Materialize Authority (MATAU)" on page 482
- "Materialize Authority List (MATAL)" on page 486
- "Materialize Authorized Objects (MATAUOBJ)" on page 492
- "Materialize Authorized Users (MATAUU)" on page 502
- "Materialize User Profile (MATUP)" on page 941
- "Materialize User Profile Pointers from ID (MATUPID)" on page 949
- "Modify Invocation Authority Attributes (MODINVAU)" on page 968
- "Test Authority (TESTAU)" on page 1165
- "Test Extended Authorities (TESTEAU)" on page 1178
- "Test User List Authority (TESTULA)" on page 1195

Process and Thread Management

- "Materialize Process Activation Groups (MATPRAGP)" on page 739
- "Materialize Process Attributes (MATPRATR)" on page 742
- "PCO Pointer (PCOPTR)" on page 1021
- "Retrieve Thread Count (RETTHCNT)" on page 1055
- "Retrieve Thread Identifier (RETTHID)" on page 1057
- "Return PCO Pointer (PCOPTR2)" on page 1064
- "Test Pending Interrupts (TESTINTR)" on page 1185
- "Wait On Time (WAITTIME)" on page 1256

Storage and Resource Management

-
- "Ensure Object (ENSOBJ)" on page 372
- "Materialize Access Group Attributes (MATAGAT)" on page 453
- "Materialize Resource Management Data (MATRMD)" on page 833
- "Set Access State (SETACST)" on page 1095
- "Yield (YIELD)" on page 1261

Dump Space Management

-
- "Materialize Dump Space (MATDMPS)" on page 552

Journal Management

- "Materialize Journal Port Attributes (MATJPAT)" on page 603
- "Materialize Journal Space Attributes (MATJSAT)" on page 612

Machine Observation

-
- "Materialize Instruction Attributes (MATINAT)" on page 566
- "Materialize System Object (MATSOBJ)" on page 921
- "Test Performance Data Collection (TESTPDC)" on page 1187
- "Test Temporary Object (TESTTOBJ)" on page 1192

Machine Interface Support Functions

-
- "Generate Universal Unique Identifier (GENUUID)" on page 406
- "Materialize Machine Attributes (MATMATR)" on page 619
- "Materialize Machine Data (MATMDATA)" on page 693
- "Materialize Machine Information (MATMIF)" on page 697

---

iSeries(TM) Exceptions

- General exception information
- All exceptions (page)
- 02 Access Group (page 1291)
- 04 Access State (page)
- 06 Addressing (page)
- 08 Argument/Parameter (page)
- 0A Authorization (page)
- 0C Computation (page)
- 0E Context Operation (page)
- 10 Damage Encountered (page)
- 12 Data Base Management (page)
- 14 Event Management (page)
- 16 Exception Management (page)
- 18 Independent Index (page)
- 1A Lock State (page 1296)
- 1C Machine-Dependent (page)
- 1E Machine Observation (page)
- 20 Machine Support (page)
- 22 Object Access (page)
- 24 Pointer Specification (page)
- 26 Process Management (page)
- 28 Process/Thread State (page)
- 2A Program Creation (page)
- 2C Program Execution (page)
- 2E Resource Control Limit (page)
- 30 Journal (page)
- 32 Scalar Specification (page)
- 34 Source/Sink Management (page)
- 36 Space Management (page)
- 38 Template Specification (page)
- 3A Wait Time-Out (page)
- 3C Service (page)
- 3E Commitment Control (page)
- 40 Dump Space Management (page)
- 44 Protection Violation (page)
- 45 Heap Space (page)
- 46 Queue Space (page)
- 48 Kernel Environment (page)
- 4A Java(TM) (page)
- 4C Signals Management (page)
- 4E Handle-Based Object (page)
- Error Conditions

The following is a list of all exceptions in alphabetic and numeric order by group. The subtypes within each group are in numeric order.

02 Access Group

01 Object ineligible for access group

EX0201.htm″>Object ineligible for access group

04 Access State

01 Access state specification invalid

EX0401.htm″>Access state specification invalid

06 Addressing

01 Space addressing violation

02 Boundary alignment

03 Range

04 External data object not found

05 Invalid space reference

07 Unsupported space use

09 Space address is not a teraspace address

EX0609.htm″>Space address is not a teraspace address

08 Argument/Parameter

01 Parameter reference violation

02 Argument list length violation

03 Argument list length modification violation

0A Authorization

01 Unauthorized for operation

02 Privileged instruction

03 Attempt to grant/retract authority state to an object that is not authorized

04 Special authorization required

05 Create/modify user profile beyond level of authorization

06 Grant/retract authority invalid

08 Unable to generate UID/GID

09 Duplicate UID/GID specified

0A ID index not available

0B Cannot transfer to new owner or primary group

EX0A0B.htm″>Cannot transfer to new owner or primary group

0C Computation

01 Conversion

02 Decimal data

03 Decimal point alignment

04 Edit digit count

05 Edit mask syntax

06 Floating-point overflow

07 Floating-point underflow

08 Length conformance

09 Floating-point invalid operand

0A Size

0B Zero divide

0C Invalid floating-point conversion

0D Floating-point inexact result

0E Floating-point zero divide

0F Master key not defined

10 Weak key not valid

11 Key parity invalid

12 Invalid extended character data

13 Invalid extended character operation

14 Invalid compressed data

15 Date boundary overflow

16 Data format error

17 Data value error

18 Date boundary underflow

19 Space pointer operands do not point to the same space object

20 Substitution character used

21 Source verification error

22 Unpaired shift control

23 Source information error

24 Receiver buffer length exceeded

EX0C24.htm″>Receiver buffer length exceeded

0E Context Operation

01 Duplicate object identification

02 Object ineligible for context

EX0E02.htm″>Object ineligible for context

10 Damage Encountered

02 Machine context damage state

04 System object damage state

05 Authority verification terminated due to damaged object

44 Partial system object damage

EX1044.htm″>Partial system object damage

12 Data Base Management

01 Conversion mapping error

02 Key mapping error

03 Cursor not set

04 Data space entry limit exceeded

05 Data space entry already locked

06 Data space entry not found

07 Data space index invalid

08 Incomplete key description

09 Duplicate key value in existing data space entry

0A End of path

0B Duplicate key value detected

0D No entries locked, or entries locked to a different thread

0F Duplicate key value in uncommitted data space entry

11 Compare key mapping error

12 Incomplete compare key description

13 Invalid mapping template

14 Invalid selection template

15 Data space not addressed by index

16 Data space not addressed by cursor

17 Key value changed since set cursor

18 Invalid key value modification

19 Invalid rule option

1A Data space entry size exceeded

1B Logical data space entry size limit exceeded

1C Key size limit exceeded

1D Logical key size limit exceeded

21 Unable to maintain a unique key data space index

25 Invalid data base operation

26 Data space index with invalid floating-point field build termination

27 Data space index key with invalid floating-point field

30 Specified data space entry rejected

31 New data space entry image rejected

32 Join value changed

33 Data space index with non-user exit selection routine build termination

34 Non-user exit selection routine failure

36 No mapping code specified

37 Operation not valid with join cursor

38 Derived field operation error

39 Derived field operation error during build index

40 Invalid entry definition table

41 ISV parameter value in runtime data pointer array not correct

42 Non-unique fanout on unique join

43 Invalid data definitional attributes template (DDAT)

45 Invalid frogger array template

46 Invalid global literal list

47 Invalid per-data space selection template

48 Re-create/replace cursor

49 No control block for shared derivation

50 Referential constraint violation

51 Object unavailable for referential constraint enforcement

52 Invalid constraint state

53 Invalid journal for referential constraint enforcement

54 Cursor must be activated under secondary commit cycle

55 Operation conflict with data space constraints

56 Constraint structure damage

57 Check constraint violation

63 Data space index templates are invalid

64 Data space index templates/keys are invalid

65 Copy index not temporary (invalid - nonenforcing)

66 Data spaces for copy index and permanent index do not match

9D Maximum number of unique encoded vector index values exceeded

EX129D.htm″>Maximum number of unique encoded vector index values exceeded

14 Event Management

01 Duplicate event monitor

02 Event monitor not present

03 Machine event requires specification of a compare value

04 Wait on event attempted while masked

05 Disable timer event monitor invalid

06 Signal timer event monitor invalid

07 Wait on event not allowed in kernel mode

07.htm″>Wait on event not allowed in kernel mode

16 Exception Management

01 Exception description status invalid

02 Exception state of thread invalid

03 Invalid invocation address

04 Retry/resume invalid

05 No inquiry message found for reply message

06 Invalid control action specified

EX1606.htm″>Invalid control action specified

18 Independent Index

01 Duplicate key argument in index

1A Lock State

01 Invalid lock state

02 Lock request not grantable

03 Invalid unlock request

04 Invalid object lock transfer request

05 Invalid space location unlocked

EX1A05.htm″>Invalid space location unlocked

1C Machine-Dependent

01 Machine-dependent request invalid

02 Program limitation exceeded

03 Machine storage limit exceeded

04 Object storage limit exceeded

05 System address range limit exceeded

06 Machine lock limit exceeded

07 Modify main storage pool controls invalid

08 Requested function not valid

09 Auxiliary storage pool number invalid

0A Service processor unable to process request

0B Program not valid for machine

0C Attribute cannot be modified from a secondary partition

0D Requested data collection function not valid

0E IASP resources exceeded

0F Software license management function failed

10 Failure due to logical partitioning action

11 Independent ASP varied offL

EX1C11.htm″>Independent ASP varied offL

1E Machine Observation

01 Program not observable

02 Invocation not found

03 Invalid D-code instruction

04 DBGINT error

05 DBGINT error on operation

EX1E05.htm″>DBGINT error on operation

20 Machine Support

01 Diagnose

02 Machine check

03 Function check

04 Invalid OS/400(R) PASE System Call

EX2004.htm″>Invalid OS/400(R) PASE System Call

22 Object Access

01 Object not found

02 Object destroyed

03 Object suspended

04 Object not eligible for operation

05 Object not available to process

06 Object not eligible for destruction

07 Authority verification terminated due to destroyed object

08 Object compressed

0A Program not eligible for operation

0B Object not available

0D Object has partial transactions

EX220D.htm″>Object has partial transactions

24 Pointer Specification

01 Pointer does not exist

02 Pointer type invalid

03 Pointer addressing invalid object type

04 Pointer not resolved

EX2404.htm″>Pointer not resolved

26 Process Management

02 Queue full

EX2602.htm″>Queue full

28 Process/Thread State

01 Process ineligible for operation

02 Process control space not associated with a process

03 User profile UID/GID invalid

04 Thread handle not associated with an active thread

0A Process attribute modification invalid

EX280A.htm″>Process attribute modification invalid

2A Program Creation

01 Program header invalid

02 ODT syntax error

03 ODT relational error

04 Operation code invalid

05 Invalid op code extender field

06 Invalid operand type

07 Invalid operand attribute

08 Invalid operand value range

09 Invalid branch target operand

0A Invalid operand length

0B Invalid number of operands

0C Invalid operand ODT reference

0D Reserved bits are not zero

10 Automatic storage for procedure exceeds maximum

11 Machine automatic storage exceeds maximum

12 Data type or length of initial value not valid

13 Exceeded internal limit on number of temporary segments

14 Static data initialized to address of automatic data

15 Initial value for static data not valid

16 Number of procedures exceeds maximum allowed

17 Type table entry not valid

18 Alias table entry not valid

19 Size of constants exceeds maximum

1A Procedure size exceeds maximum

1B Instruction stream not valid

1C Size of literals exceeds maximum

1D Dictionary entry not valid

1E Level of machine interface not supported on target release

1F Size of dictionary exceeds maximum

20 Internal machine operation not valid

21 Size of internal binding table exceeds maximum

22 Size of internal label table exceeds maximum

23 Size of internal symbolic register table exceeds maximum

24 Size of internal computation table exceeds maximum

28 Size of internal basic block table exceeds maximum

29 Size of internal successor arc table exceeds maximum

2A Size of internal register table exceeds maximum

2B Size of internal late bound offset table exceeds maximum

2C Invalid function prototype component entry

5E An error was detected in a static storage definition or initialization

5F Overlapping initializations not valid

60 Dictionary ID is not valid

61 Binding specification value not valid

62 Copyright component value not valid

63 Module limitation exceeded

A0 Attempt to delete part that may not be deleted

B0 Object list referential extension not valid

B1 Symbol resolution list referential extension not valid

B2 Service program export list referential extension not valid

B3 Secondary associated spaces list referential extension not valid

B4 Program limitation exceeded

B5 Observable information necessary for retranslation not encapsulated

B6 Procedure order list referential extension not valid

C0 Attempt to delete part that may not be deleted

C1 An attempt was made to delete a required module part

EX2AC1.htm″>An attempt was made to delete a required module part

2C Program Execution

01 Return instruction invalid

02 Return point invalid

04 Branch target invalid

05 Activation in use by invocation

06 Instruction cancellation

07 Instruction termination

08 Branch target defined by label pointer not valid

10 Process object destroyed

11 Process object access invalid

12 Activation group access violation

13 Activation group not found

14 Activation group in use

15 Invalid operation for program

16 Program activation not found

17 Default activation group not destroyed

18 Invalid source invocation

19 Invalid origin invocation

1A Invocation offset outside range of current stack

1B Invocation not eligible for operation

1C Instruction not valid for invocation type

1D Automatic storage overflow

1E Activation access violation

1F Program signature violation

20 Static storage overflow

21 Program import invalid

22 Data reference invalid

23 Imported object invalid

24 Activation group export conflict

25 Import not found

26 Invalid activation group

27 Unresolved import

28 Activation group directory error

29 Caller parameter mask does not match procedure parameter mask

2A Caller parameter mask does not match imported procedure parameter mask

2B Invalid Storage Model

EX2C2B.htm″>Invalid Storage Model

2E Resource Control Limit

01 User profile storage limit exceeded

02 Security audit journal failure

EX2E02.htm″>Security audit journal failure

30 Journal

01 Apply journal changes failure

02 Entry not journaled

03 Maximum objects through a journal port limit exceeded

04 Invalid journal space

05 Invalid selection/transaction list entry

06 Journal space not at a recoverable boundary

07 Journal ID not unique

08 Object already being journaled

09 Transaction list limit reached

0A Data space index currently journaled

0B Data space index currently in force mode

0C Underlying data space not journaled to same journal

0E File ID not available

10 Unrecoverable system managed access path protection failure

11 Journal violation

12 Object entry invalid

20 Remote journal operation error

EX3020.htm″>Remote journal operation error

32 Scalar Specification

01 Scalar type invalid

02 Scalar attributes invalid

03 Scalar value invalid

EX3203.htm″>Scalar value invalid

34 Source/Sink Management

01 Source/sink configuration invalid

02 Source/sink physical address invalid

03 Source/sink object state invalid

04 Source/sink resource not available

41 Invalid starting verb index was specified

42 System pointer area was tagged and non-zero

43 Invalid open in list of operations

44 Unsupported stream operation

45 Incorrect length specified

EX3445.htm″>Incorrect length specified

36 Space Management

01 Space extension/truncation

02 Invalid space modification

EX3602.htm″>Invalid space modification

38 Template Specification

01 Template value invalid

02 Template size invalid

03 Materialization length invalid

04 Invalid mutex

EX3804.htm″>Invalid mutex

3A Wait Time-Out

01 Dequeue time-out

02 Lock time-out

03 Event time-out

04 Space location lock wait time-out

05 Object location lock wait time-out

EX3A05.htm″>Object location lock wait time-out

3C Service

01 Invalid service session state

02 Unable to start service session

EX3C02.htm″>Unable to start service session

3E Commitment Control

01 Invalid commit block status change

02 Commit block must be decommitted

03 Commit block is attached to process

04 Commit blocks control uncommitted changes

05 Operation not valid on commit block in prepared state

06 Commitment control resource limit exceeded

07 Operation not valid on commit block in decommit only state

08 Object under commitment control being journaled incorrectly

09 Reconstruct of commit block environment failure

10 Operation not valid under commitment control

11 Process has attempted to attach too many commit blocks

12 Objects under commitment control

13 Commit block not journaled

14 Errors during decommit

15 Object ineligible for commitment control

16 Object ineligible for removal from commitment control

EX3E16.htm″>Object ineligible for removal from commitment control

40 Dump Space Management

01 Dump data space size limit exceeded

02 Invalid dump data insertion

03 Invalid dump space modification

04 Invalid dump data retrieval

04.htm″>Invalid dump data retrieval

44 Protection Violation

01 Object domain or hardware storage protection violation

02 Literal values cannot be changed

03 Cannot Change Contents of Protected Context

EX4403.htm″>Cannot Change Contents of Protected Context

45 Heap Space

01 Invalid heap identifier

02 Invalid request

03 Heap space full

04 Invalid size request

05 Heap space destroyed

06 Invalid heap space condition

07 Invalid mark identifier

46 Queue Space

01 Queue space not associated with the process

02 Queue space cannot be modified

03 Message reference index does not identify a valid message

04 Queue space not eligible for destruction

EX4604.htm″>Queue space not eligible for destruction

48 Kernel Environment

01 Function code out of range

02 Kernel environment already active

03 Kernel environment not initialized

04 Kernel environment not active

05 MI wait in kernel mode terminated

EX4805.htm″>MI wait in kernel mode terminated

4A Java

A0 Generic Java exception

A1 Java class format error

A2 Java verify error

A3 Java archive file error

A4 Java invalid native method return

A5 Java virtual machine terminated

A6 Java native interface unattached caller

A7 Java Stand-Alone Program Creation Error

EX4AA7.htm″>Java Stand-Alone Program Creation Error

4C Signals Management

01 Asynchronous signal terminated MI wait

02 Signal controls not initialized

03 Asynchronous signal received

03.htm″>Asynchronous signal received

4E Handle-Based Object

01 Invalid handle

02 Invalid handle type

03 Handle-based primitive exists

- Storage Synchronization

Appendix A. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

```
IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY  10594-1785
U.S.A.
```

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

```
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan
```

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:
Advanced 36
Advanced Function Printing
Advanced Peer-to-Peer Networking
AFP
AIX
AS/400
COBOL/400
CUA
DB2
DB2 Universal Database
Distributed Relational Database Architecture
Domino
DPI
DRDA
eServer
GDDM
IBM
Integrated Language Environment
Intelligent Printer Data Stream
IPDS
iSeries

Lotus Notes
MVS
Netfinity
Net.Data
NetView
Notes
OfficeVision
Operating System/2
Operating System/400
OS/2
OS/400
PartnerWorld
PowerPC
PrintManager
Print Services Facility
RISC System/6000
RPG/400
RS/6000
SAA
SecureWay
System/36
System/370
System/38
System/390
VisualAge
WebSphere
xSeries

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Appendix B. Terms and conditions for downloading and printing publications

Permissions for the use of the publications you have selected for download are granted subject to the following terms and conditions and your indication of acceptance thereof.

Personal Use: You may reproduce these Publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these Publications, or any portion thereof, without the express consent of IBM(R).

Commercial Use: You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations. IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

By downloading or printing a publication from this site, you have indicated your agreement with these terms and conditions.

Appendix C. Code disclaimer information

This document contains programming examples.

IBM(R) grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

All sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

All programs contained herein are provided to you ″AS IS″ without any warranties of any kind. The implied warranties of non-infringement, merchantability and fitness for a particular purpose are expressly disclaimed.

Printed in USA