Burroughs

B 1000 Systems
SDL/UPL

REFERENCE MANUAL

(RELATIVE TO MARK 10.0 RELEASE)

PRICED ITEM

Printed in U.S.A. July 1982 1137833

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to TIO West Documentation, Burroughs Corporation, 1300 John Reed Court, City of Industry, California 91745, U.S.A.

LIST OF EFFECTIVE PAGES

Page	Issue
Title	Original
ii	Original
iii	Original
iv	Blank
v thru xii	Original
xiii thru xiv	Original
1-1 thru 1-5	Original
1-6	Blank
2-1 thru 2-11	Original
2-12	Blank
3-1 thru 3-7	Original
3-8	Blank
4-1 thru 4-45	Original
4-46	Blank
5-1 thru 5-3	Original
5-4	Blank
6-1 thru 6-14	Original
7-1 thru 7-12	Original
8-1 thru 8-13	Original
8-14	Blank
9-1 thru 9-236	Original
10-1 thru 10-17	Original
10-18	Blank
11-1 thru 11-3	Original
11-4	Blank
A-1 thru A-2	Original
B-1 thru B-18	Original
C-1 thru C-58	Original
D-1 thru D-27	Original
D-28	Blank
1 thru 16	Original

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

PREFACE

This manual describes the SDL/UPL programming language. The manual is divided into 11 sections and 4 appendixes. Each is briefly described as follows:

Section	Contents
1	INTRODUCTION Provides a brief introduction to the SDL/UPL language and compiler. Lists the related documents and describes the notation and syntax conventions used in this manual.
2	FUNDAMENTALS OF THE LANGUAGE Defines the valid characters, identifiers, literals, constants, and data types allowed in an SDL/UPL source program. The use of comments in an SDL/UPL source program is also described.
3	STRUCTURE OF AN SDL/UPL PROGRAM Describes the structure of an SDL/UPL source program.
4	DECLARATIONS Describes the use of declarations in an SDL/UPL source program. This includes simple, structured, dynamic, paged array, file, switch_file, and reference declarations.
5	DEFINES Describes the use of defines in an SDL/UPL source program.
6	EXPRESSIONS Describes the use of expressions in an SDL/UPL source program. This includes unary, arithmetic, relational, logical, conditional expression, and replacement operators and their order of precedence.
7	PROCEDURES Describes the use of procedures in an SDL/UPL source program. This includes the use of parameters and the type option in procedures, procedure invocations, and forward procedure declarations.
8	STATEMENTS Describes the valid statements allowed in an SDL/UPL program.
9	VERBS Describes the use of the verbs in an SDL/UPL source program.
10	COMPILER OPTIONS AND PASSES Describes the options, conditional compilation modes, and the passes of the SDL/UPL compiler
11	HOW TO WRITE AN SDL/UPL PROGRAM Describes the writing rules and form of an SDL/UPL program. Also, example programs are provided.

Section	**Contents**
A	SPECIAL AND RESERVED WORDS Lists the SDL and UPL reserved and special words.
B	THE SDL ENVIRONMENT Describes the SDL program environment.
C	SDL/UPL SYNTAX REFERENCE GUIDE Contains all the railroad syntax diagrams for all the SDL/UPL declarations and verbs.
D	GLOSSARY OF COMMONLY USED TERMS AND ACRONYMS Describes the terms and acronyms used throughout this manual.

SECTION 1
INTRODUCTION

The Burroughs B 1000 computer system is a small, general-purpose computer system. The B 1000 differs from other computer systems in that it is dynamically microprogrammable and is designed to support many independent special-purpose machine architectures, rather than one general-purpose architecture.

Each particular machine architecture is realized on a microprogrammable B 1000 processor by means of multiprogrammed interpreters. The general philosophy of the B 1000 computer system is that each language that runs on the machine has its own interpreter. For example, the B 1000 computer system can be a "COBOL machine," a "FORTRAN machine," a "BASIC machine," an "RPG machine," and so forth.

To permit this flexibility, a language (along with its interpreter) was designed to be used for implementation of the Master Control Program (MCP), the various compilers, the Network Definition Language (NDL), the Data Management System (DMSII), and all the utility programs. This language is called the Software Development Language (SDL).

SDL is tailored to the B 1000 computer system and provides access to all machine features. Use of some of the SDL verbs requires that the programmer have intricate "state of the art" knowledge of the B 1000 system. These verbs are used exclusively for system software development. Therefore, the User Programming Language (UPL) was created to provide the flexibility of SDL without any of the potentially dangerous verbs. Throughout the remainder of this manual the term "SDL/UPL" is used to imply both the SDL and UPL compilers and languages. The terms "SDL" and "UPL" are used to refer to the respective compiler or language.

UPL is a high-level, problem-oriented language that allows sophisticated computer programs to be written with relative ease. The flexibility of UPL makes it a powerful programming tool for the system user as well as the system designer. The language can increase programmer productivity and can make the solution of complex problems easier. The resultant software reflects this increased productivity.

RELATED DOCUMENTS

The following documents are referenced in this document:

B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982.

B 1800/B 1700 Systems System Software Operation Guide, Volume 2, form number 1108966.

B 1000 Systems SORT Reference Manual, form number 1090594.

NOTATION CONVENTIONS

Left and Right Broken Brackets (< >)

Left and right broken bracket characters are used to enclose letters and digits which are supplied by the user. The letters and digits can represent a variable, a number, a file name, or a command.

Example:

```
<job #>AX<command>
```

AT SIGN (@)

The at sign (@) character is used to enclose hexadecimal information.

Example:

```
@F3@ is the hexadecimal representation of the EBCDIC
character 3.
```

The @ character is also used to enclose binary or hexadecimal information when the initial @ character is followed by a (1) or (4), respectively.

Examples:

```
@(1)11110011@ is the binary representation of the EBCDIC
character 3.

@(4)F3@ is the hexadecimal representation of the EBCDIC
character 3.
```

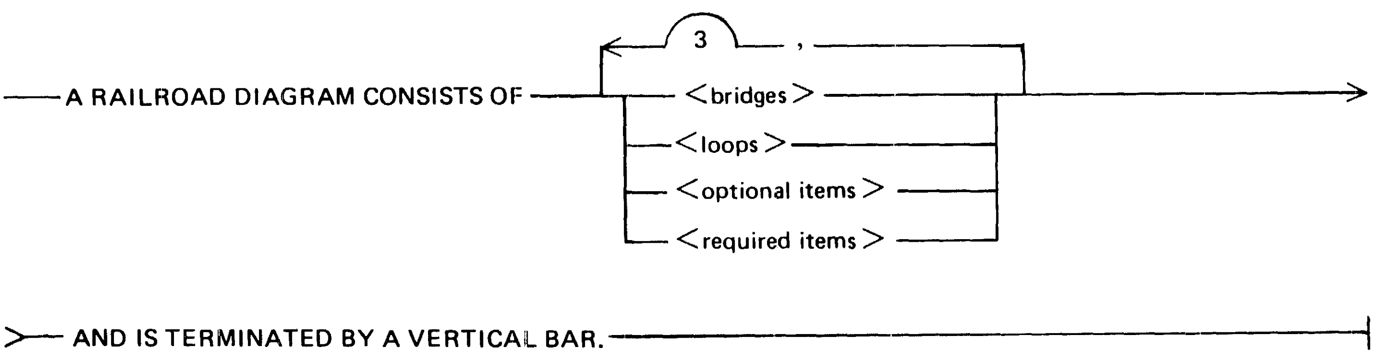
SYNTAX CONVENTIONS

Railroad diagrams show how syntactically valid statements can be constructed.

Traversing a railroad diagram from left to right, or in the direction of the arrow heads, and adhering to the limits illustrated by bridges will produce a syntactically valid statement. Continuation from one line of a diagram to another is represented by a right arrow (→) appearing at the end of the current line and beginning of the next line. The complete syntax diagram is terminated by a vertical bar (|).

Items contained in broken brackets (< >) are syntactic variables which are further defined, or require the user to supply the requested information.

Upper-case items must appear literally. Minimum abbreviations of upper-case items are underlined.



The following syntactically valid statements may be constructed from the above diagram:

A RAILROAD DIAGRAM CONSISTS OF <bridges> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional items> AND IS TERMINATED BY A VERTICAL BAR.

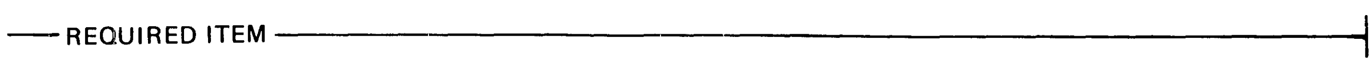
A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional items>, <required items>, <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

Required Items

No alternate path through the railroad diagram exists for required items or required punctuation.

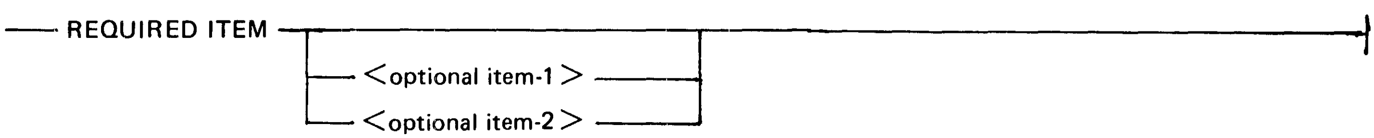
Example:



Optional Items

Items shown as a vertical list indicate that the user must make a choice of the items specified. An empty path through the list allows the optional item to be absent.

Example:



The following valid statements may be constructed from the preceding diagram:

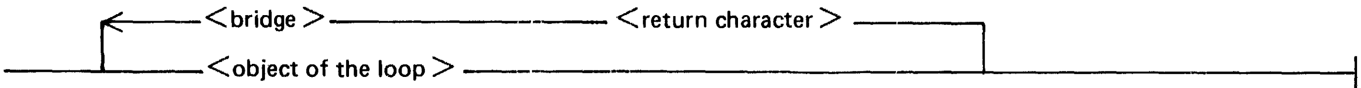
REQUIRED ITEM

REQUIRED ITEM <optional item-1>

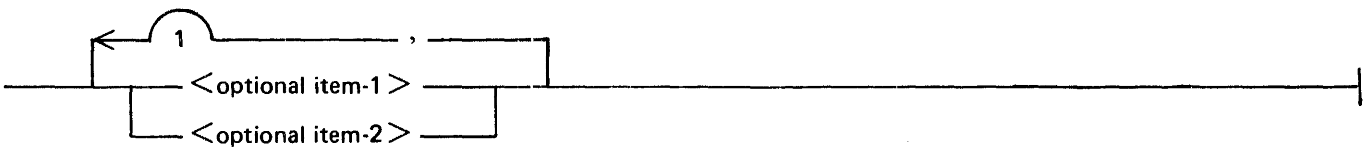
REQUIRED ITEM <optional item-2>

Loops

A loop is a recurrent path through a railroad diagram and has the following general format:



Example:



The following statements can be constructed from the railroad diagram in the example.

<optional item-1>

<optional item-2>

<optional item-1>,<optional item-1>

<optional item-1>,<optional item-2>

<optional item-2>,<optional item-1>

<optional item-2>,<optional item-2>

A <loop> must be traversed in the direction of the arrow heads, and the limits specified by bridges cannot be exceeded.

Bridges

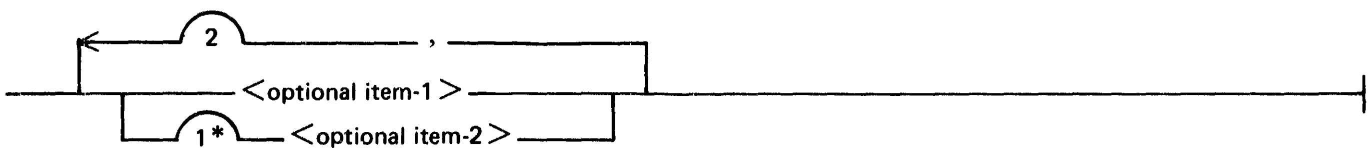
A bridge indicates the minimum or maximum number of times a path may be traversed in a railroad diagram.

There are two forms of <bridges>.

n is an integer which specifies the maximum number of times the path may be traversed.

n is an integer which specifies the minimum number of times the path must be traversed.

Example:



The loop may be traversed a maximum of two times; however, the path for <optional item-2> must be traversed at least one time.

The following statements can be constructed from the railroad diagram in the example.

<optional item-2>

<optional item-1>,<optional item-2>

<optional item-2>,<optional item-2>,<optional item-1>

<optional item-2>,<optional item-2>,<optional item-2>

SECTION 2
FUNDAMENTALS OF THE LANGUAGE

The SDL/UPL language is a problem-solving oriented language which requires a series of functions and constructs that differ significantly from most other problem-oriented languages. The following is a list of the most common differences.

- Powerful bit and character-string functions.
- Binary-only arithmetic functions.
- No JUMP or GO TO instruction.
- Re-entrant programs (B 1000 computer system characteristic)
- Recursive procedures (subroutines).
- Scope of identifiers contained within procedures.
- Dynamic storage allocation for identifiers at execution time.

All programs that are written in the SDL/UPL source language must be processed by the SDL/UPL compiler. The SDL/UPL compiler transforms the source statements into a virtual machine form called the S-Machine language. Refer to Appendix B for a description of the S-Machine. The S-Machine language is then executed interpretively by a set of micro-instruction routines (firmware).

SDL/UPL PROPERTIES

An SDL/UPL program has a distinct pattern or format that specifies the relative locations of the two statement types, declaration and executable. Declaration statements provide the information that is needed to allocate storage or link together various elements of a program. Executable statements specify the functions or transformations that occur upon the contents in storage.

Statements are composed of symbols that, in turn, are composed of letters, digits, and special characters. Symbol strings are called operands, operators, or control functions. The SDL/UPL syntax is concerned with the correct creation of symbol strings and the relative placement of the strings to form declarative and executable statements.

SDL/UPL PROGRAM FORMAT

SDL/UPL programs are segmented into logical subdivisions called procedures. Each procedure begins with a head statement and terminates with an end statement. Procedures have a definite relationship to other procedures within a program, either side-by-side (parallel) or subordinate (nested). This ordering inherently defines the scope of each procedure and the range over which a procedure can call (or be called by) another procedure.

All procedures have a rigid internal structure. The procedure structure is as follows: the data declarations appear first, all nested procedures appear second, and all executable statements appear last. Nested procedure structures must be identical.

SDL/UPL SOURCE FILE RECORD FORMAT

The format of a source file record to the SDL/UPL compiler consists of the following information.

1. Columns 1 through 72 contain the SDL/UPL statements, declarations, or comments.
2. Columns 73 through 80 contain the sequence number of the source file record.

CHARACTER SET

The following characters are allowed in an SDL/UPL source program.

Letters	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Digits	0 1 2 3 4 5 6 7 8 9
Special Characters	+ – * / = < > : ; ( ) [ ] @ # " % ? $ & . - _ | (space)

The collating sequence for letters, digits, and special characters is based on standard EBCDIC representation.

Table 2-1 shows the function of each symbol that is used in an SDL/UPL program.

Table 2-1. Use of Punctuation Symbols in an SDL/UPL Program

Symbol	Definition	Use
_	Underscore	Concatenation within identifier names
.	Period	Concatenation within identifier names for record structures and field selection
,	Comma	Separator for items
;	Semicolon	Delimiter for statements
(	Left parenthesis	Enclose parameter lists and array subscripts (leading)
)	Right parenthesis	Enclose parameter lists and array subscripts (trailing)
"	Quotation mark	Left and right character string delimiter
#	Number sign	Left and right define text string delimiter
	Space or blank	Identifier delimiter
@	At sign	Bit string delimiter
!	Exclamation mark	Assignment or replacement (delete left)
:=	Colon, equal sign	Assignment or replacement (delete left)

Table 2-1. Use of Punctuation Symbols in an SDL/UPL Program (Cont)

Symbol	Definition	Use
::=	Colon, colon, equal sign	Replacement (delete right) operator symbol
%	Percent sign	Remainder of record is a comment
/*	Virgule, asterisk	Beginning of comment
*/	Asterisk, virgule	End of comment
$	Dollar sign	In position one of a source record, indicates a compiler control option
&	Ampersand	In position one of a source record, indicates a conditional source record inclusion control statement
[	Left Bracket	Enclose the record key and cospatial fields of records (leading)
]	Right Bracket	Enclose the record key and cospatial fields of records (trailing)
+	Plus sign	Addition operator
-	Minus sign	Subtraction operator
/	Virgule	Division operator
*	Asterisk	Multiplication operator
=	Equal sign	Equal relation operator
/=	Virgule, equal sign	Not equal relation operator
>	Greater than sign	Greater than relation operator
>=	Greater than, equal sign	Greater than or equal relation operator
<	Less than sign	Less than relation operator
<=	Less than, equal sign	Less than or equal relation operator

IDENTIFIERS

An identifier is a defined name which is a symbolic representation for a location in memory. Identifiers are often called data names and field names in other computer languages.

An identifier must begin with a letter.

An identifier cannot contain blanks.

An identifier can contain a maximum of 64 characters.

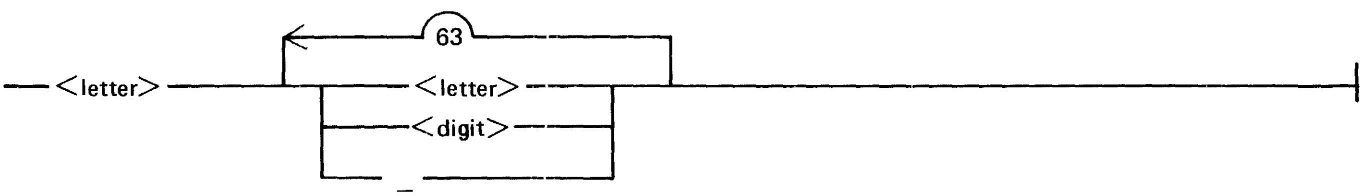
Reserved words cannot be used as identifiers. Reserved words in SDL/UPL are listed in Appendix A.

Special words are used for segment and DO-group identifiers and do not lose their special significance in SDL/UPL. Special words lose their special significance when defined as identifiers. When defined at lexicographic (lexic) level 0, they lose their significance throughout the entire program. Defined at any higher level, they lose their significance within the procedure in which they are defined. Special words in SDL/UPL are listed in Appendix A.

Identifiers must contain exactly the same letters in the same case (upper or lower) to be identical. The identifier THIS_ONE differs from the identifier this_one.

The railroad syntax diagrams of both SDL and UPL are presented.

SDL and UPL Syntax:



Syntax Semantics:

letter
This field can be any valid letter defined in the SDL/UPL character set.

digit
This field can be any valid digit defined in the SDL/UPL character set.

_
The underscore (_) character can be used to concatenate groups of letters and digits.

ARRAY IDENTIFIERS

An array identifier is a defined name which is a symbolic representation for a number of contiguous locations in memory that correspond to each element within the array.

SDL and UPL Syntax:

——— <identifier> (<subscript>) ———|

Semantics:

identifier
This field can be any valid SDL/UPL identifier and specifies the name of the array.

subscript
This field can be any valid SDL/UPL expression that returns a binary value and specifies the element within the array. The elements in an array begin with 0 and end with n − 1, where n is the total number of elements declared for the array.

Examples:

```
A(10)        % References element 10 of array identifier A.

ARRAY (0)    % References element 0 of array identifier ARRAY.
```

DATA TYPES

All data used in an SDL/UPL program must be declared and allocated storage space. There are four different data types allowed in an SDL/UPL program: FIXED, BIT, CHARACTER, or RECORD. These data types, or a combination of them, are used to define all data used in an SDL/UPL program.

FIXED

The FIXED data field is a signed, 24-bit field. The leftmost bit is the sign bit. If the sign bit is 1, the field is negative. If the sign bit is 0, the field is positive. Negative numbers are represented in two's complement notation.

Examples:

```
 +1 = @(1)000000000000000000000001@ = @(4)000001@

 -1 = @(1)111111111111111111111111@ = @(4)FFFFFF@

+10 = @(1)000000000000000000001010@ = @(4)00000A@

-10 = @(1)111111111111111111110110@ = @(4)FFFFF6@
```

The numbers 1 and 4 enclosed in parentheses denote binary and hexadecimal representations, respectively.

The FIXED data field is the basic computational form in the SDL/UPL program. The values for a FIXED data field can range from −(2 EXP 23) to (2 EXP 23)−1 [−8,388,608 to 8,388,607]. Arithmetic overflow is ignored.

BIT

A BIT data field can be any variable-length string of bits. The maximum length for a string of bits in an SDL/UPL program is 65,535 bits.

When used in arithmetic computations, bit data is treated as a 24-bit, unsigned number. Values can range between 0 and (2 EXP 24)−1 (16,777,215). If a BIT data field is the target field of an arithmetic computation and the field is greater than 24 bits in length, only the rightmost 24 bits are used. The resulting leftmost bit is not interpreted as a sign bit. Prior to any arithmetic operation on BIT data fields, the data is right-aligned and zero-filled on the left.

Examples:

```
@(1)111000@
@(1)1@
@(1)000000000000000000001111@ = @(4)00000F@ = 15
```

CHARACTER

A CHARACTER data field can contain any variable-length string of characters. Each variable-length string is represented by an 8-bit EBCDIC code. The maximum number of characters allowed in a CHARACTER data field is 8191 characters.

If a CHARACTER data field is used in an arithmetic operation, the following must be noted.

- The binary value of the CHARACTER data field is used. Blank characters are represented as @(1)01000000@ or @(4)40@ which is not the same as the binary representation of the number zero.

- Only the rightmost 24 bits of a CHARACTER data field are used in an arithmetic operation.

The results of CHARACTER-to-CHARACTER operations are aligned on the left and the blank fill or truncate operations are aligned on the right. CHARACTER-to-BIT or CHARACTER-to-FIXED arithmetic operations align the data on the right and the zero-fill or truncate operations align the data on the left.

Most input/output operations treat their operands as CHARACTER data and thus follow the rules of CHARACTER-to-CHARACTER operations.

RECORD

A record is an addressing template. Declaration of the record causes no data space to be allocated. The declaration only establishes an addressing scheme in the scope of the declaration.

Specifying a record declaration is done by using the RECORD keyword in the declarations. Refer to RECORDS DECLARATIONS in Section 5 for a complete description of declaring a record.

CONVERSION BETWEEN DATA TYPES

The conversion verbs CONVERT, BINARY, and DECIMAL transform data from one data type to another. When the value of a number is to be written in a readable form, the DECIMAL verb should be used.

VALUES AND ADDRESSES OF VARIABLES

An identifier is a symbolic reference to the value at a memory address associated with a type and length attribute. A reference to an identifier is always a reference to the value at the address associated with the identifier when the identifier appears to the right of an assignment or replacement operator within an expression.

When an identifier appears to the left of an assignment or replacement operator, the reference is to the address of the identifier. To force references to the value rather than the address of an identifier, enclose the identifier within parentheses.

The identifier is considered a target identifier because its memory address receives the value generated when the expression on the right of that operator is evaluated.

Literals, operator expressions, and keyword expressions cannot be used as target identifiers because they generate values rather than addresses.

The verbs which can be used as target identifiers are SUBBIT and SUBSTR.

LITERALS

A literal is an item of data which contains a value identical to the characters being described. There are three classes of literals in an SDL/UPL source program: numeric, bit strings, and character strings.

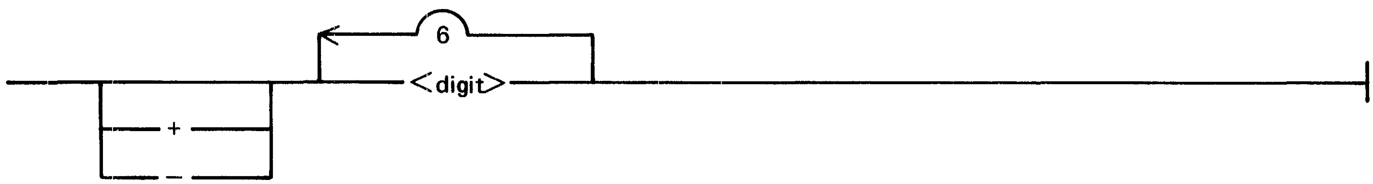
Numeric Literal

A numeric literal represents an integer value and cannot be the designation identifier of an assignment operation.

Numeric literals cannot exceed a value of 16,777,215.

Imbedded blank characters are not allowed.

SDL and UPL Syntax:



Syntax Semantics:

\+
The plus sign (+) character makes the numeric literal a positive number.

–
The minus sign (–) character makes the numeric literal a negative number.

digit
This field can be any valid digit that is in the SDL/UPL character set.

Examples:

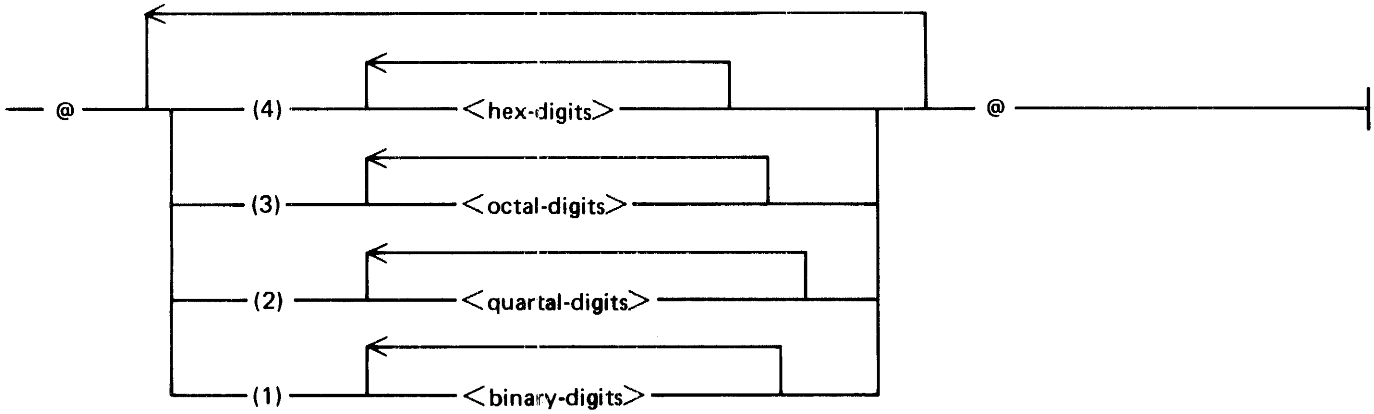
```
12345
807
-27
+32
```

Bit-String Literal

A bit-string literal can be a combination of hexadecimal, octal, quartal, and binary digits. The bit-string literal is delimited by the at sign (@) character. A number from 1 to 4 enclosed within parentheses designates the base integer system.

Imbedded blank characters are not allowed.

SDL and UPL Syntax:



Syntax Semantics:

@
The at sign (@) character is used to delimit the bit string.

(4), (3), (2), (1)
The numbers enclosed within parentheses specify that the following digits are hexadecimal (hex), octal, quartal, and binary digits, respectively.

hex-digits
This field can be any of the hexadecimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, or F.

octal-digits
This field can be any of the octal digits 0, 1, 2, 3, 4, 5, 6, or 7.

quartal-digits
This field can be any of the quartal digits 0, 1, 2, or 3.

binary-digits
This field can be either of the binary digits 0 or 1.

Examples:

```
@(4)BEEF@          % Hexadecimal bit string and value equals 48879.

@CAFE@             % Hexadecimal bit string and value equals 51966.

@(3)7654@          % Octal bit string and the value equals 4012.

@(2)3210@          % Quartal bit string and the value equals 228.

@(1)10101010@      % Binary bit string and the value equals 170.
```

Character-String Literal

A character-string literal can be any combination of EBCDIC characters enclosed within quotation mark (") characters. Character-string literals must be completely described to the SDL/UPL compiler in one source record.

Character-string literals can be concatenated with others by using the CAT operator to build larger character-string literals. The maximum length of a character-string literal is 256 characters.

Example of an invalid split of a character-string literal:

```
Record n  : " A B C
Record n+1:    X Y Z "

n represents the relative record number of a source file record.
```

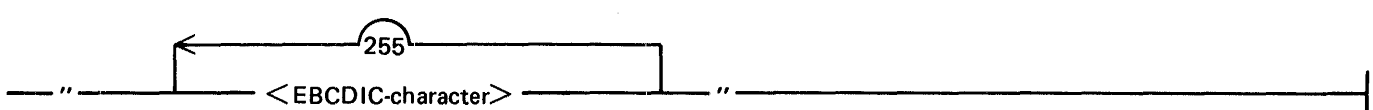
Example of a valid split of a character-string literal:

```
Record n  : " A B C "
Record n+1:     CAT " X Y Z "

n represents the relative record number of a source file record.
```

The string concatenator operator CAT must be used to enter long character literals. If the CAT operator is used, the compiler treats the literal as a single string.

Two adjacent quotation mark (") characters must be used to include a quotation mark (") character within the character string.

SDL and UPL Syntax:



Syntax Semantics:

"

The quotation mark (") character is used to delimit the character string.

EBCDIC-character

This field can be any valid character defined in the SDL/UPL character set.

Examples:

```
" "" "       yields   "
"ABC""DEF"   yields   ABC"DEF
```

MISCELLANEOUS CONSTANTS

The following keywords represent values that are compiled into the SDL/UPL program as constants.

```
HEX_SEQUENCE_NUMBER
SEQUENCE_NUMBER
TODAYS_DATE
```

HEX_SEQUENCE_NUMBER

The constant HEX_SEQUENCE_NUMBER represents a bit string of eight (hex) digits. This bit string is the sequence field, columns 73-80 of the source image, in the source file in which the HEX_SEQUENCE_NUMBER keyword appears. If this sequence field is blank, HEX_SEQUENCE_NUMBER is @00000000@.

Example:

If the current source image line sequence number is 12753000, then on this line:

```
HEX_SEQUENCE_NUMBER = @12753000@
```

SEQUENCE_NUMBER

The constant SEQUENCE_NUMBER represents a character string of eight characters. This character string is the sequence field, columns 73-80 of the source image, in the source file in which the SEQUENCE_NUMBER keyword appears. If this sequence field is blank, SEQUENCE_NUMBER is 00000000.

Example:

If the current source image line sequence number is 12753000, then on this line:

```
SEQUENCE_NUMBER = 12753000
```

TODAYS_DATE

The constant TODAYS_DATE represents the date and time of compilation of the SDL/UPL program. It is the same as the date and time which appears at the top of the SDL/UPL program listing. The TODAYS_DATE constant is a character string with the format MM/DD/YY hh:mm, where MM represents the month, DD represents the day, YY represents the year, hh represents the hour, and mm represents the minutes of the compile.

COMMENTS

Comments are allowed in SDL/UPL programs and have no effect on program execution. There are two forms of comments. These are:

1. The enclosed comment, which must be enclosed within the virgule (/) and asterisk (*) character pair.
2. The end-of-record comment, which is preceded by the percent sign (%) character.

Enclosed Comment

The enclosed comment begins with a virgule-asterisk (/*) character pair and ends with an asterisk-virgule (*/) character pair. When the virgule-asterisk (/*) pair is encountered, the SDL/UPL compiler continues scanning the current source-image record until the asterisk-virgule (*/) pair is found. If the current source-image record does not have the ending asterisk-virgule (*/) character pair, the SDL/UPL compiler scans the next and subsequent source file records until the ending asterisk-virgule (*/) is found.

SDL and UPL Syntax:

```
/* <comment-text> */
```

Syntax Semantics:

comment-text
: This field can contain any comment that the programmer desires to include for documentation purposes.

Example:

```
CODE /* This is an example of an enclosed comment text.  This
        text begins with the virgule-asterisk pair and ends with
        the asterisk-virgule pair.  */ STATEMENT;
```

End-of-Record Comment

The end-of-record comment begins with the percent sign (%) character and continues to the end of the source file record. The SDL/UPL compiler discontinues scanning of a source image record when a percent sign (%) character is encountered. If a percent sign (%) character is contained within comment text delimited by the virgule-asterisk (/*) and the asterisk-virgule (*/) character pairs, the percent sign (%) character is treated as a part of the comment text. The SDL/UPL compiler then continues scanning for the ending asterisk-virgule (*/) character pair.

The percent sign (%) character is not treated as an end-of-record indicator if it is imbedded in a quoted character string. For example, "% THIS IS A PERCENT SIGN".

SDL and UPL Syntax:

```
% <comment-text>
```

Syntax Semantics:

%
: The percent sign (%) character indicates that the remainder of the source image is <comment-text>.

comment-text
: This field can contain any comment that the programmer desires to include for documentation purposes.

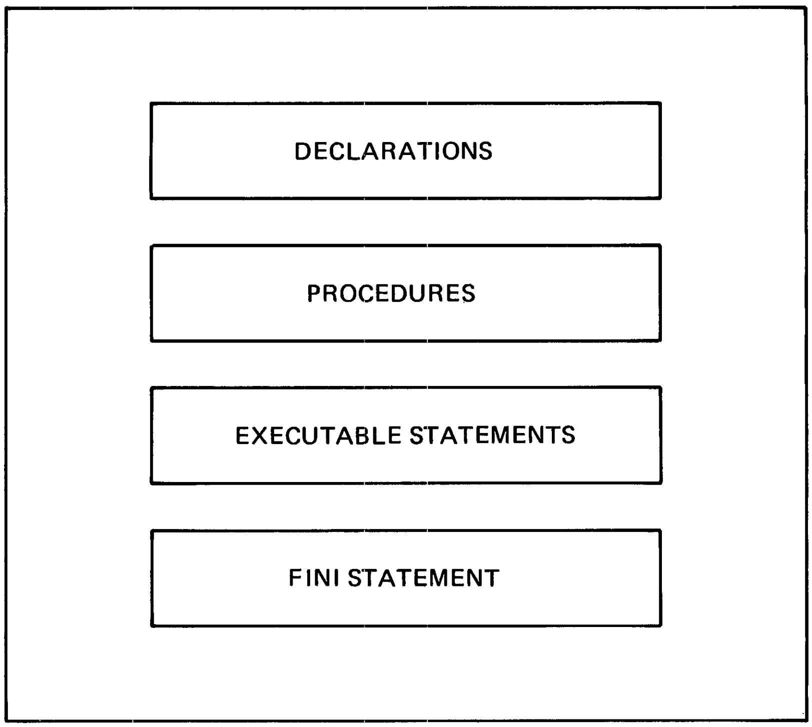
Example:

```
CODE STATEMENT;    % This is the end-of-record comment text.
```

SECTION 3
STRUCTURE OF AN SDL/UPL PROGRAM

The structure of an SDL/UPL source program includes four kinds of statements in this order: declarations, procedures, executable statements, and a FINI statement (or end-of-file record).

Figure 3-1 illustrates the basic structure of the SDL/UPL source program.



G18297

Figure 3-1. Basic Structure of the SDL/UPL Source Program

An SDL/UPL program can have procedures within a procedure. A procedure within a procedure is called a "nested" procedure and has the same basic structure as the structure of an SDL/UPL program. Nested procedures consist of declarations, procedures (optional) and executable statements. A nested procedure begins with PROCEDURE <procedure name> and ends with END <procedure name>. Refer to Section 7 for a complete description of procedures in an SDL/UPL source program.

LEXICOGRAPHIC LEVEL

A lexicographic (lexic) level is a compile-time relationship of each procedure to the outer level of the program. The outer level is referred to as lexic level 0 (zero). All other procedures are nested within lexic level 0. They are assigned a lexic level number which represents their depth of nesting from lexic level 0. Figure 3-2 shows the relationship of procedures and their associated lexic level number.

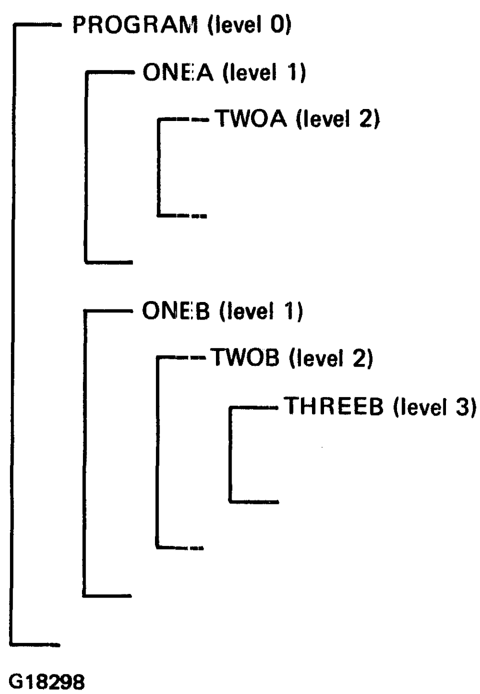


Figure 3-2. Relationship of Procedures and Lexic Level Number

Procedures ONEA and ONEB are at lexic level 1, procedures TWOA and TWOB are at lexic level 2, and procedure THREEB is at lexic level 3.

The maximum lexic level is 15. Nested procedures cannot exceed 15 levels in depth. There is no limit to the number of procedures that can occur on any level or in any procedure.

Declaring a procedure (procedure identifier) must not be confused with the procedure itself. The procedure identifier exists at some lexic level and specifies that a procedure is beginning with the next source statement. The next source statement exists within the procedure and is one lexic level number higher than the procedure identifier. This separation of the procedure identifier from its procedure has significance in the scope of a procedure. Figure 3-3 is a coding example showing procedures nested within other procedures in an SDL/UPL source program.

```
DECLARE A1, A2, A3, A4;

PROCEDURE B;
   DECLARE B1, B2, B3;
   PROCEDURE C;
      DECLARE C1, C2, C3;
      Executable Statements
   END C;
   PROCEDURE D;
      Executable Statements
   END D;
Executable Statements
END B;

PROCEDURE E;
   DECLARE E1, E2;
   PROCEDURE F;
      DECLARE F1, F2, F3;
      PROCEDURE G;
         DECLARE G1, G2;
         Executable Statements
      END G;
      PROCEDURE H;
         Executable Statements
      END H;
   Executable Statements
   END F;
   PROCEDURE J;
      DECLARE J1, J2;
      PROCEDURE K;
         DECLARE K1, K2;
         Executable Statements
      END K;
   Executable Statements
   END J;
   Executable Statements
END E;

Executable Statements

FINI;
```

Figure 3-3. Example Showing Procedures Nested within Procedures

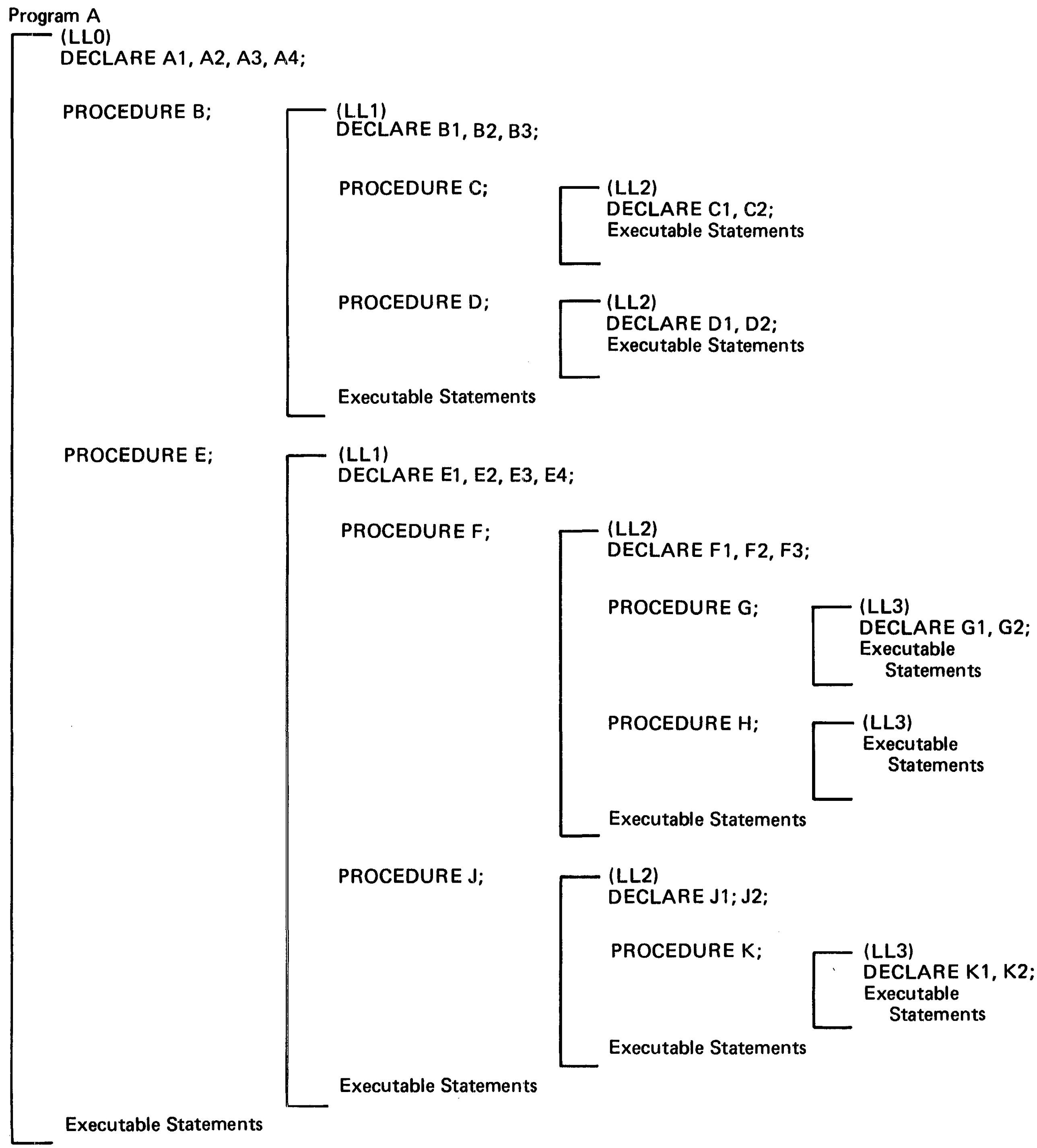
SCOPE OF PROCEDURES AND IDENTIFIERS

The scope of a procedure, determined at compile time by the SDL/UPL compiler, is the range within a program over which an identifier or procedure identifier can be referenced. The scope of an identifier is a direct result of the lexic level of procedures and of the storage allocation techniques used by the SDL/UPL compiler. The scope of an identifier is that portion of the SDL/UPL program which can reference the identifier. The scope of a global identifier is all the nested procedures and statements, exclusive of any nested procedures and statements that declare the same identifier. Nested procedures and statements are procedures and statements embedded within the procedure such that the different hierarchical (lexic) levels can be performed or accessed recursively.

The scope of an identifier within a procedure is that procedure exclusive of any nested procedures within the procedure that declares the same identifier.

The format of procedures ensures that only those statements contained within the procedure or in global procedures (procedures with lower lexic level numbers) are within the scope of the procedure. Executable statements in a procedure can reference identifiers and procedure identifiers that are declared in that procedure.

Figure 3-4 illustrates the scope of a sample program.



G18299

Figure 3-4. Procedure Nesting

In Figure 3-4, the procedure identifier is assigned the lexic level number of the encompassing procedure. The procedure itself is assigned the next higher lexic level number. LL1, LL2, and LL3 represent lexic level numbers 1, 2, and 3, respectively. Procedure D is at lexic level 2 while the procedure identifier D is at lexic level 1.

The executable statements in lexic level 0 can reference procedure identifiers B and E, but not procedure identifiers C, D, F, G, H, J, and K. They cannot because procedures B and E have not been invoked and procedure identifiers C, D, F, G, H, J and K are not defined.

The executable statements in procedure B can reference procedure identifiers C and D because procedure identifiers C and D become available when procedure B is invoked.

The executable statements in procedure B can also reference any identifiers or procedures that are declared on lexic level 0. This implies that procedure B can invoke itself. All procedures are recursive. Any difficulties encountered with duplicate identifiers within a nested procedure are resolved by the allocation of new space for the most recent occurrence of the duplicate identifier.

The executable statements in procedures G and H can reference identifiers within procedures E and F. Executable statements in procedure K can reference identifiers within procedures E and J.

Several procedures can have the same lexic level number by occurring at the same depth from lexic level 0. The relationships that can exist between such procedures depend upon the relationship of the nested procedures in which they appear.

Procedures that have a common procedure (one lexic level number lower) can invoke each other. Procedures that do not have this attribute cannot invoke each other.

The following are conditions for inclusion of an identifier within the scope of a procedure.

- The procedure identifier itself.
- Procedures declared in the procedure, but not their nested procedures. Thus, in Figure 3-4, procedure identifier F is within the scope of procedure E while procedure G is not.
- Any procedure (and its nested procedures) whose procedure identifier is declared at the same lexic level and within the same procedure as its own identifier.
- The procedure in which its own procedure identifier is declared.

The known scope is limited by the requirement that an identifier must be declared before it can be referenced. Thus, in Figure 3-4, procedure B cannot reference procedure identifier E, although procedure E can reference procedure identifier B. A FORWARD procedure declaration removes this restriction. Refer to the Section 7 for a complete description of FORWARD procedures.

The scope of an identifier includes all procedures which can reference the identifier. An identifier can be either a data name or a procedure name. In Figure 3-4, executable statements in procedure C can reference procedure identifier B. Procedure identifier C is within the scope of procedure B. Executable statements in procedure C can invoke procedure identifier B. Executable statements in procedure C can reference identifiers B1, B2, and B3.

Table 3-1 is used in conjunction with Figure 3-4 and shows the relationship between the scope of a procedure and the invoking procedure.

Table 3-1. Relationship of Scope and Invoking Procedures

		Invoking Procedure								
	A	B	C	D	E	F	G	H	J	K
Procedure Identifier	B	*	*	*	*	*	*	*	*	*
	C		*	*	*					
	D		*	*	*					
	E	*	*	*	*	*	*	*	*	*
	F					*	*	*	*	*
	G						*	*	*	
	H						*	*	*	
	J					*	*	*	*	*
	K									*

To find the scope of a procedure in Table 3-1, find the procedure identifier in the first column. The horizontal rows to the right of each procedure identifier indicate the procedures in its scope. The procedures which can be invoked by a given procedure are indicated by an asterisk in the vertical columns below the invoking procedure identifier.

SECTION 4
DECLARATIONS

This section describes data, record, file, and switch-file declarations that can be specified in an SDL/UPL program.

The data declaration specifies simple, overlay (remap), structured, reference, dynamic, and paged-array data items.

The record declaration specifies a data structure which does not allocate memory space and is used in conjunction with the data declaration.

The file declaration describes a file to be used by an SDL/UPL program.

The switch-file declaration, which specifies a group of files that can be used as files, is referenced by a subscript.

DATA DECLARATIONS STATEMENT

The DECLARE statement specifies simple, overlay (remap), structured, reference, dynamic, and paged-array data items. The fundamental data types that can be declared are BIT, CHARACTER, and FIXED. Additionally, the programmer can define a combination of these data types in a RECORD declaration, and subsequently use that RECORD structure as a data type in declaration clauses.

Any error in a declaration statement causes the SDL/UPL compiler to ignore all other declarations that occur within the same statement and beyond the point of error. Everything between the error and the end-of-statement token (;) is ignored.

The SDL/UPL compiler generates more efficient code when all declare clauses are in a single DECLARE statement.

All of a procedure's declaration statements must appear before any executable statements.

Spaces between the data type keywords BIT and CHARACTER and their parenthesized sizes are optional.

Example:

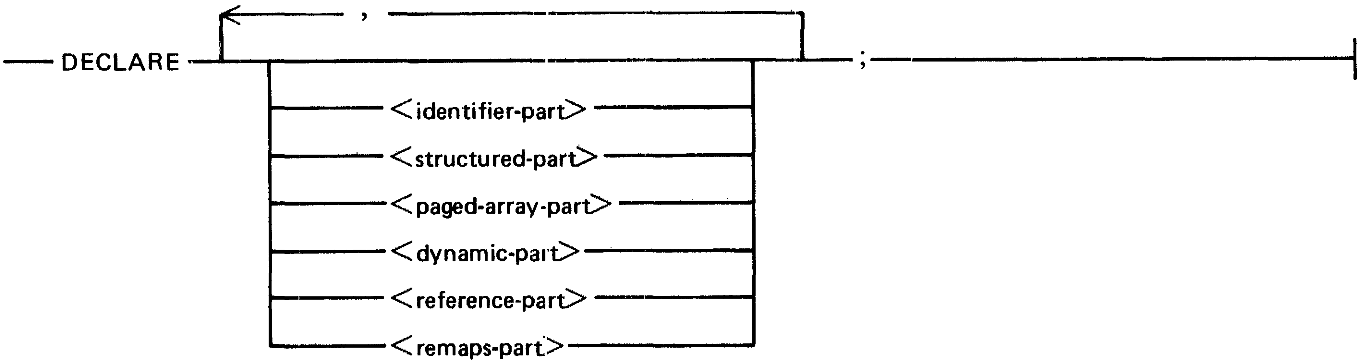
```
"CHARACTER(10)" and "CHARACTER (10)"
```

Spaces are also optional between an array identifier and its subscript.

Examples:

```
DECLARE  A            FIXED,
         B            CHARACTER,
         (C,E,F(5))   FIXED,
         H(5)         CHARACTER(6);
```

SDL and UPL Syntax:



Syntax Semantics:

identifier-part
    Refer to identifier-part in this section.

structured-part
    Refer to structured-part in this section.

paged-array-part
    Refer to paged-array-part in this section.

dynamic-part
    Refer to dynamic-part in this section.

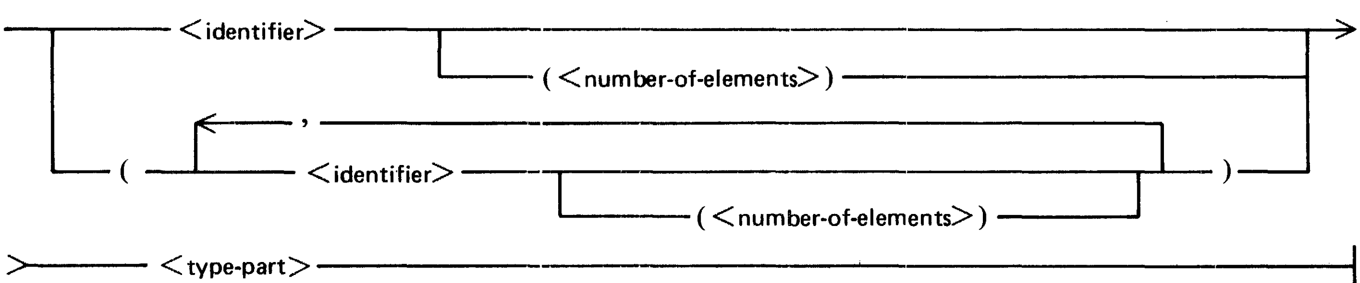
reference-part
    Refer to reference-part in this section.

remaps-part
    Refer to remaps-part in this section.

identifier-part

The syntax and semantics of the identifier-part in the DECLARE statement are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

identifier
    This field can be any valid SDL/UPL identifier and specifies the name of a data item or array.

number-of-elements
    This field specifies the size of an array and can be any valid SDL/UPL number, identifier, or expression that returns a binary value.

    An SDL/UPL array is a group of memory locations associated with a single identifier. All elements of an array are identical in structure. Individual array elements are referenced by using a subscript with the array identifier.

    Any identifier followed by a number in parentheses names an array.

    Array subscripts are zero-relative. For example, the first element of array ARRAY is ARRAY(0). Valid subscripts for a 5-element array are 0, 1, 2, 3, and 4. If the subscript is not between 0 and n-1 inclusive, where n is the declared number of elements in the array, an invalid subscript error is generated and the program is terminated by the MCP.

    The maximum number of elements that can be specified for an array is 65,535. The maximum length of the array is 65,535 bits (8191 characters).

type-part
    Refer to type-part in this section.

structured-part

The structured-part of the DECLARE statement allows the programmer to specify data items in logical groups. The maximum number of data items allowed in a single structure is 198. The keywords DUMMY and FILLER are included in this count. Any attempt to declare a larger structure causes a table overflow error at compile time.

The size of a structure can be specified in the data type of its 01-level identifier. When no data type is specified, the compiler assigns a structure size equal to the aggregate length in bits of all subfields of the structure.

The two following structures cause identical structures to be generated. Both DECLARE statements generate an implied 3-bit filler.

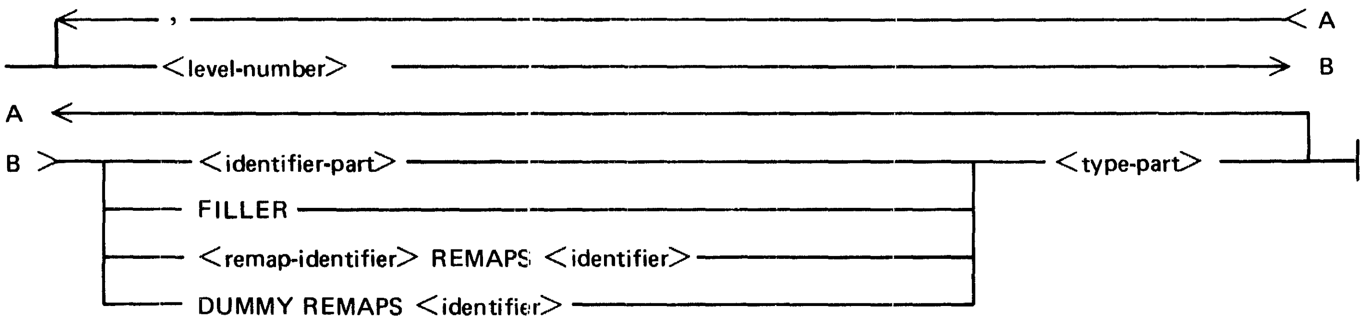
Example:

```
DECLARE   01 A      CHARACTER(4),
             02 B   FIXED,
             02 C   BIT (5);

DECLARE   01 A     CHARACTER,
             02    B FIXED,
             02    C BIT (5);
```

Data items that are specified with level numbers also called "structured data" can be remapped. If the REMAP keyword appears on a level greater than 1, the remap is restricted. In this case, the right-hand identifier must be the last data item in the same structure on the same level as the lefthand identifier that is to remap it. If the previous data item was declared with the REMAPS keyword, the right-hand identifier can refer to the original declaration of the memory space.

The syntax, semantics, and some examples of the structured-part in the DECLARE statement are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

level-number
: This field can be any valid SDL/UPL 2-digit integer and specifies the level of the structure. <level-number> can range from 01 to 99.

identifier-part
: Refer to identifier-part in this section.

type-part
: Refer to type-part in this section.

FILLER
: The keyword FILLER designates the memory areas which the program does not reference. The FILLER keyword can be used on any level specified by <level-number> which is greater than 01. If the FILLER keyword is the last item in a structure and its parent field specified a length, it can be omitted. The SDL/UPL compiler supplies an implied filler. An item's parent identifier is the field which the item subdivides. The parent identifier must have a lower level number than its subdividing item.

remap-identifier
: This field can be any valid SDL/UPL identifier and specifies an alternative identifier for the same memory space declared by <identifier>.

REMAPS
: The keyword REMAPS causes memory space specified by <identifier> to be named <remap-identifier>. When the REMAPS keyword appears on a structure with <level-number> greater than 01, <identifier> must be the last data item declared in the same structure having a level number of <identifier> that is equal to the level number of <remap-identifier>. Also, <remap-identifier> must be the last data item declared in the same structure with equal level numbers unless the last data item is also declared with the REMAPS keyword.

DUMMY
: The keyword DUMMY can be substituted for <remap-identifier>, but a data descriptor is not generated. The DUMMY keyword can be used only in conjunction with the REMAPS keyword. The DUMMY keyword eliminates the need to declare redundant identifiers.

The DUMMY keyword cannot be used to remap another DUMMY keyword.

If the DUMMY keyword is specified, the subordinate structure must have at least one identifier that is not the FILLER keyword.

Examples:

```
DECLARE  01 A                     BIT (160),
            02 B                  BIT (60),
            02 FILLER             BIT (20),
            02 C                  CHARACTER (10),

         01 AA REMAPS A           CHARACTER (20),
            02 BB                 BIT (80),
            02 CC                 BIT (80),
            02 BMAP REMAPS BB     CHARACTER (10),

         01 DUMMY REMAPS A        BIT (160),
            02 BBB (6)            FIXED,
            02 FILLER             BIT (16);   % This FILLER is optional.
```

paged-array-part

The paged-array-part in the DECLARE statement allows SDL/UPL programs to use the B 1000 system's dynamic memory facility. This facility allows the amount of memory to vary depending on how much is actually used and can be set at execution time with the MEMORY program attribute. Refer to the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982 for a complete description of the MEMORY program attribute. The amount of dynamic memory allocated can also be set by specifying the $ DYNAMICSIZE compiler option.

The SDL/UPL compiler automatically allocates dynamic memory sufficient for one page of each paged array declared. From this, the programmer must allocate enough additional dynamic memory based on the knowledge of how many pages are actually used at any one time. If the amount of dynamic memory is not enough at execution time, the following program abort message is displayed on the Operator Display Terminal (ODT):

```
SDL PAGED ARRAY HANDLER COULDN'T OBTAIN <number> BITS.
--INSUFFICIENT DYNAMIC MEMORY- RERUN WITH ME=<number>
```

The syntax, semantics, and an example of the paged-array-part in the DECLARE statement are described as follows:

SDL and UPL Syntax:

```
-------- PAGED (<elements-per-page>) <identifier> ---------------------->

 >----- (<number-of-elements>)  <type-part> ---------------------------|
```

Syntax Semantics:

PAGED
The keyword PAGED causes the array specified by <identifier> to be segmented. Paged-arrays cannot be indexed, a part of a structure, or remapped.

elements-per-page
This field specifies the number of elements of the array specified by <identifier> to be contained in an overlayable data segment. It can be any valid SDL/UPL number or expression that returns a binary value. <elements-per-page> must be one of the following values: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, or 32768.

identifier
This field can be any valid SDL/UPL identifier and specifies the name of the array to be segmented into pages.

number-of-elements
This field specifies the number of elements in the array and can be any valid SDL/UPL number or expression that returns a binary value. <number-of-elements> can range from 1 to 65,535, inclusive. <number-of-elements> can be increased up to 16,777,215 by using the GROW verb. Refer to Section 9 for complete information on the syntax, semantics, and function of the GROW verb.

type-part
Refer to type-part in this section.

Example:

```
DECLARE PAGED (64) A (4096) BIT (1);  % Array identifier A is a
                                      % segmented array with 64
                                      % elements per segment, and a
                                      % total of 4096 elements, each
                                      % one bit long.
```

dynamic-part

The syntax, semantics, and examples of the dynamic-part in the DECLARE statement are described as follows:

SDL and UPL Syntax:

```
—— DYNAMIC <identifier-part> ——┬———————————————————┬————————————|
                               └—— <remap-part> ———┘
```

Syntax Semantics:

DYNAMIC
The keyword DYNAMIC allows the array length of <identifier> or <number-of-elements> to be determined at the time the procedure is entered.

The keyword DYNAMIC can be specified only in a procedure. Any variables specified must have been previously declared and initialized.

The keyword DYNAMIC cannot be specified on lexic level 0.

No length checks are made when a dynamic identifier is remapped. Any remapping of a dynamic identifier generates an advisory message from the SDL/UPL compiler.

identifier-part
Refer to identifier-part in this section.

remap-part
Refer to remap-part in this section.

Example 1:

```
PROCEDURE ABC;                          % The length of identifier X is
 DECLARE DYNAMIC X   BIT (A);           % determined by the value of
    .                                   % identifier A.
    .
    .
END ABC;
```

Example 2:

```
PROCEDURE XYZ;                          % The number of elements in
 DECLARE DYNAMIC A (B)  BIT (10);       % array A is determined by the
    .                                   % value of identifier B.
    .
    .
END XYZ;
```

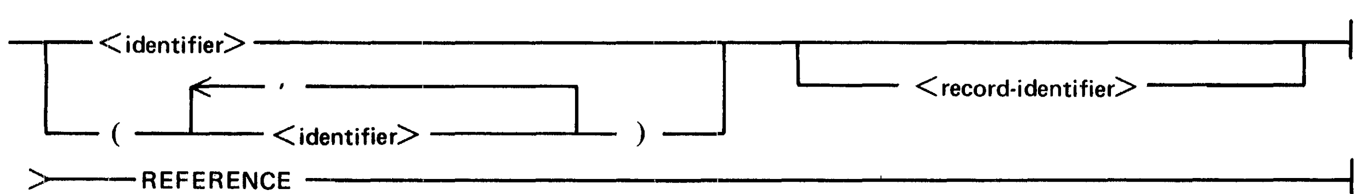
Example 3:

```
DECLARE X FIXED;                        % The value of identifier X
PROCEDURE NESTED;                       % determines the number of
 DECLARE DYNAMIC ABC(X) FIXED;          % elements in array ABC.
    .
    .
    .
END NESTED;
X := 10;
NESTED;
STOP;
FINI;
```

reference-part

The syntax, semantics, and an example of the reference-part in the DECLARE statement are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

identifier

This field can be any valid SDL/UPL identifier and specifies the name of the reference identifier.

record-identifier

This field can be any valid SDL/UPL record identifier and specifies a RECORD reference identifier. RECORD reference identifiers are assigned with a REFER verb and can be written in other statements as though they were structure identifiers. For example, a RECORD reference identifier can have field qualifiers attached with the period (.) notation. Such an access divides the current memory areas described by the reference identifier according to the record declaration.

Example:

```
DECLARE DR DESCRIPTOR REFERENCE;        % Identifier X is assigned
                                        % to bits 108 through 124
REFER DR TO SUBBIT (MYAREA, 100, 48);   % of the bit string MYAREA.
X := DR.LEN;
```

All restrictions which apply to normal reference identifiers are applicable to RECORD reference identifiers. RECORD reference identifiers cannot be specified in the REDUCE verb.

REFERENCE

The keyword REFERENCE causes <identifier> to be a reference identifier. Reference identifiers are used as pointers to data without allocating memory space. Since reference identifiers are pointers, the REMAPS keyword cannot have a data type equal to REFERENCE. A reference identifier is bound to another identifier by using the REFER verb.

Generally, reference identifiers are used as a scanning tool. The reference identifier is bound to an identifier that has a data type equal to CHARACTER or an expression that returns a value with a data type equal to CHARACTER. The REFER verb is used to bind a reference identifier to an identifier. The REDUCE verb is used on the reference identifier to obtain the desired character string. Refer to Section 9 for information concerning the REFER and REDUCE verbs.

Example:

```
DECLARE  A  REFERENCE,          % The reference identifier A is
         B  CHARACTER (20);     % bound to identifier B.
REFER A TO B;
```

remaps-part

The syntax, semantics, and some examples of the remaps-part in the DECLARE statement are described as follows:

SDL Syntax:

```
---<remap-identifier> REMAPS ---+--- <identifier> ---+--- <type-part> ---|
                                |                    |
                                +--- BASE -----------+
```

UPL Syntax:

```
--- <remap-identifier> REMAPS ------ <identifier> <type-part> ---------|
```

Syntax Semantics:

remap-identifier

This field can be any valid SDL/UPL identifier. It specifies the alternative name of the same memory space as <identifier>.

REMAPS

The keyword REMAPS causes the starting address of <remap-identifier> to be the same as <identifier>.

<remap-identifier> cannot be larger than <identifier>. However, it can be remapped by a smaller identifier. In that case, the SDL/UPL compiler provides implied filler bits on the unmapped rightmost bits.

Example:

```
DECLARE A                BIT (10),    % An implied 3-bit filler
        B REMAPS A       BIT(7),      % is provided for identifier
        C REMAPS B       BIT(5);      % B and an implied 5-bit
                                      % filler is provided for
                                      % identifier C.
```

There is no actual limit to the number of times a field can be remapped. <remap-identifier> can be remapped by another <remap-identifier>.

BASE

The keyword BASE is valid only for SDL programs and causes <remap-identifier> to have a starting address at the base-relative address of the program.

The keyword BASE is used as a free-standing declaration since it does not remap a previously declared identifier and is used primarily with data that is to be indexed. Refer to Section 6 for a description of indexing in SDL programs.

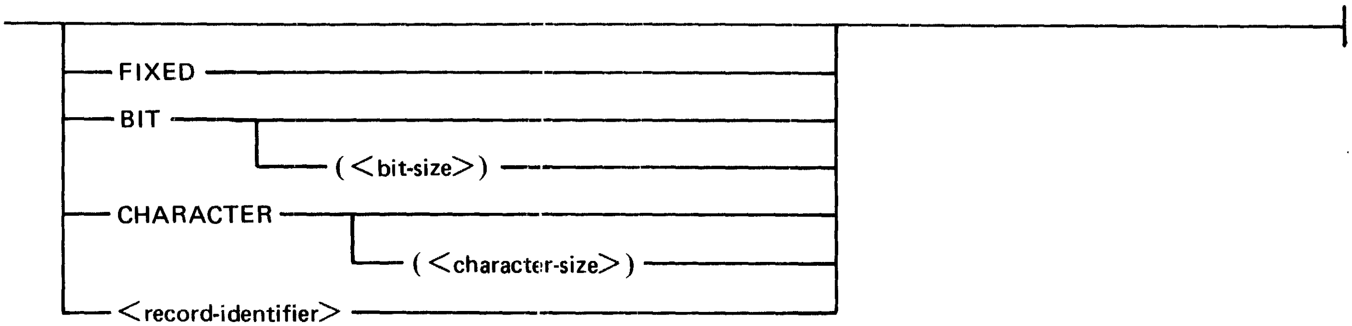
Examples:

```
DECLARE A                CHARACTER (10), % Identifier B remaps
        B REMAPS A       BIT (80),       % identifier A and identifier
        C REMAPS BASE    BIT (100);      % C has a starting address
                                         % equal to 0 (the beginning
                                         % address of the program).
```

type-part

The syntax and semantics of the type-part in the DECLARE statement are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

BIT
: The keyword BIT makes the identifier have a data type equal to BIT. A bit can have a value equal to 0 (zero) or 1. It is the smallest unit of data that can be addressed on the B 1000 computer system.

CHARACTER
: The keyword CHARACTER makes the identifier have a data type equal to CHARACTER. A character is 8-bits long and represents one of the 256 EBCDIC characters.

FIXED
: The keyword FIXED makes the identifier have a data type equal to FIXED. A fixed identifier is 24 bits long with the sign-bit in the leftmost bit position. The sign-bit is used for arithmetic calculations. A negative number is stored as the two's complement of its like positive number. Identifiers with a FIXED data type can range in value from −8,388,608 to +8,388,607, inclusive.

bit-size
: This field specifies the number of bits in <identifier> and can be any valid SDL/UPL number, identifier, or expression that returns a binary value.

character-size
: This field specifies the number of characters in <identifier> and can be any valid SDL/UPL number, identifier, or expression that returns a binary value.

record-identifier
: This field can be any valid SDL/UPL identifier and specifies the name of a record structure. Refer to Record Declarations in this section.

Array Declaration Information

Only 1-dimensional, level-structured arrays are allowed. Thus, if a group item is an array, none of its substructures can be an array. Multidimensional arrays can be created by using record structures. An array field cannot be declared with a REFERENCE data type. A multidimensional field can be define by using the RECORD REFERENCE structure.

If the 01-level identifier is an array, it is mapped as a contiguous area in memory. Subdivisions of an array are not contiguous. The following shows the way in which subdivisions of an array are mapped.

Example:

```
DECLARE 01   A(5)    BIT (48),
             02  B    FIXED,
             02  C    FIXED;
```

Figure 4-1 shows how array A and identifiers B and C are mapped in memory.

A(0)		A(1)		A(2)		A(3)		A(4)	
B(0)	C(0)	B(1)	C(1)	B(2)	C(2)	B(3)	C(3)	B(4)	C(4)

NOTE

A(0), A(1), A(2), A(3), and A(4) are all 48 bits in length. B(0), C(0), B(1), C(1), B(2), C(2), B(3), C(3), B(4), and C(4) are all 24 bits in length.

G18300

Figure 4-1. Memory Mapping of Array A and Identifier B and C

Examples of DECLARE Statements

The following are examples of DECLARE statements.

Example 1:

```
DECLARE TAGA FIXED;                  % Identifier TAGA is a signed
                                     % 24-bit binary value.  The sign
                                     % is the leftmost bit.
```

Example 2:

```
DECLARE TAGB CHARACTER;              % Identifier TAGB is of type
                                     % CHARACTER and one unit long.
                                     % CHARACTER is in 8-bit EBCDIC
                                     % format.
```

Example 3:

```
DECLARE TAGC BIT (17);               % Identifier TAGC is of type BIT
                                     % and is 17 bits long.
```

Example 4:

```
DECLARE TAGA FIXED,                  % The identifiers have the same
        TAGB CHARACTER (1),          % name, data type, and length as
        TAGC BIT (17);               % in examples 1-3, except they
                                     % are declared in one statement
                                     % with the identifiers separated
                                     % by the comma (,) character.
```

Example 5:

```
DECLARE 01 CARD     CHARACTER (80),  % An implied filler of eight
        02 INPUT    CHARACTER (72);  % characters is automatically
                                     % assigned by the SDL/UPL
                                     % compiler to expand the 02
                                     % level to its required length of
                                     % 80 characters.
```

Example 6:

```
DECLARE                              % A table of five items that
  01 TABLE_A          CHARACTER (14), % consumes 14 bytes is declared.
     02 ITEM_1        CHARACTER (6), % Each item is explicitly named
     02 ITEM_2        CHARACTER (4), % in the structure, and its type
        03 SUB_ITEM_2 FIXED,         % and length are given.  Also
     02 ITEM_3        BIT (1),       % declared is a second table of
     02 ITEM_4        FIXED,         % 200 bits. Identifier SUB_ITEM_2
     02 ITEM_5        BIT (7),       % further subdivides ITEM_2 and
  01 TABLE_B          BIT (200);     % uses the first three characters
                                     % (24 bits).  There is an implied
                                     % FILLER on the 03 level
                                     % following identifier
                                     % SUB_ITEM_2.
```

Example 7:

```
DECLARE                              % An 80-column card is declared
   CARDS              CHARACTER (80). % and then remapped as an array
   COLUMNS (80) REMAPS CARDS         % of 80 elements, each of
                      CHARACTER (1), % one character.  The card is
   01 NUM_FIELDS (40) REMAPS CARD    % remapped again as a 40-element
                      CHARACTER (1), % array, each of two characters.
      02 FIRST_NUM    CHARACTER (1), % Each 2-character array element
      02 SECOND_NUM   CHARACTER (1); % is further subdivided into
                                     % separate elements that can be
                                     % referenced.
                                     % Identifiers FIRST_NUM and
                                     % SECOND_NUM must be subscripted
                                     % when they are used. The
                                     % subscript values must range
                                     % from 0 to 39, inclusive.
```

Example 8:

```
DECLARE (ITEM1, ITEM2, ITEM3) FIXED;  % A list of identifiers is
                                      % declared, all of data type
                                      % FIXED.
```

Example 9:

```
DECLARE                               % A group item NEW_LABEL is
  01 NEW_LABEL,                       % declared and the SDL/UPL
     02 NL_1         CHARACTER (25),  % compiler assigns it a BIT
     02 NL_2 (3)     CHARACTER (25),  % data type.  The length of
        03 FILLER    CHARACTER (5),   % NEW_LABEL is equal to the
        03 FIRST     CHARACTER (10),  % sum of the bits of the 02
        03 SECOND    CHARACTER (10),  % levels that follow ((25 + 3
     02 NL_3         FIXED;           % * 25) * 8 + 24 = 824 bits).
                                      % Identifier NL_2 is an array
                                      % of three elements each 25
                                      % characters in length.  FILLER
                                      % is used to omit the naming of
                                      % an area that is never
                                      % referenced separately.
                                      % FILLER can be used as often
                                      % as required without causing
                                      % a duplicate-name syntax
                                      % error.  Identifiers FIRST and
                                      % SECOND are 3-element subarrays
                                      % of array NL_2.  They are
                                      % referenced with subscripts 0,
                                      % 1, and 2 for the first, second,
                                      % and third elements,
                                      % respectively.  Each element is
                                      % 10 characters long. Identifier
                                      % NL_3 is a FIXED, signed binary
                                      % number.
```

Example 10:

```
DECLARE  01 A,                        % Because of the explicitly
            02 A1 (20)  BIT (20),     % declared array-size specified
            02 A2 (18)  BIT (20),     % for array A1, A2, and A3,
               03 B1    BIT (15),     % identifiers A1, A2, B1, B2 and
               03 B3    BIT (5),      % A3 must all the subscripted,
            02 A3 (2)   BIT (5);      % when referenced. The length sum
                                      % of identifiers B1 and B2 must
                                      % be equal to, or less than, the
                                      % length specified for identifier
                                      % A.
```

Example 11:

```
DECLARE  01 TAGA (5) BIT (48),          % Identifier TAGA is mapped
            02 TAGB  FIXED,             % into a contiguous memory
            02 TAGC  FIXED;             % area to contain the data for
                                        % identifiers TAGB and TAGC.
                                        % TAGB and TAGC are implicit
                                        % 5-unit arrays, but are not
                                        % mapped contiguously.  They
                                        % are mapped in an alternating
                                        % manner as follows:  TAGB(0),
                                        % TAGC(0), TAGB(1), TAGC(1),
                                        % ..., TAGB(4), and TAGC(4).
```

Example 12:

```
DECLARE                                 % Identifier BIG_D_N is an array
 PAGED (1024) BIG_D_N (4096) BIT (1);   % with 4096 elements, each one
                                        % bit long.  The array is
                                        % segmented into 1024 parts. Each
                                        % part is brought into memory;
                                        % that is, paged whenever it is
                                        % addressed.  No special
                                        % statements are required to do
                                        % the paging.
```

RECORD DECLARATIONS

SDL/UPL programs have two ways of creating data structures. They are the level-structure DECLARE statement and the RECORD statement. Each statement establishes similar structures. The following are the benefits of using the RECORD statement.

- RECORD statements reduce run-time space requirements because records do not generate large name and value stacks.
- RECORD statements provide safer, simpler, and often faster access to linked data structures than do level-structured DECLARE statements.
- RECORD statements provide a method to structure paged arrays.
- RECORD statements allow arrays to be nested within structural levels.
- RECORD statements reduce the probability of error and increase programming ease by allowing structures to be described once and invoked many times.

NOTE

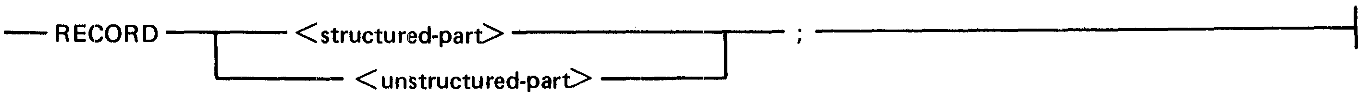
Data structures cannot be declared with a data type of REFERENCE. The RECORD REFERENCE construct must be used instead.

Building a record structure requires two statements. First, a RECORD statement must describe the memory layout of the structure. The RECORD statement essentially describes a new data type that can be used exactly as data types BIT, CHARACTER, and FIXED. Describing the RECORD structure does not allocate memory space for the structure.

Memory space is allocated for the RECORD structure when the record identifier is specified as the data type of an identifier in the DECLARE statement. Thus, a DECLARE statement is the second statement needed to invoke a RECORD structure.

The syntax and semantics of the RECORD statement are described as follows:

SDL Syntax:



UPL Syntax:

Syntax Semantics:

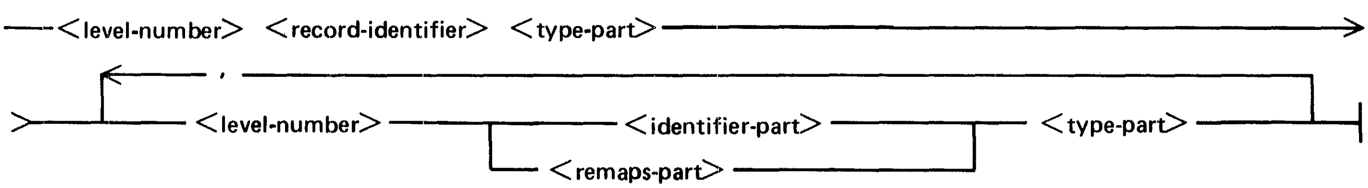
structured-part
Refer to structured-part in this section.

unstructured-part
Refer to unstructured-part in this section.

structured-part

The syntax and semantics of the structured-part in the RECORD statement are described as follows:

SDL Syntax:



Syntax Semantics:

record-identifier
This field can be any valid SDL/UPL identifier and specifies the name of the record structure.

level-number
This field can be any valid SDL/UPL number and specifies the level of the record structure. <level-number> can range from 1 to 99. The first level number for a record structure must be 01 or 1.

identifier-part
Refer to identifier-part in this section.

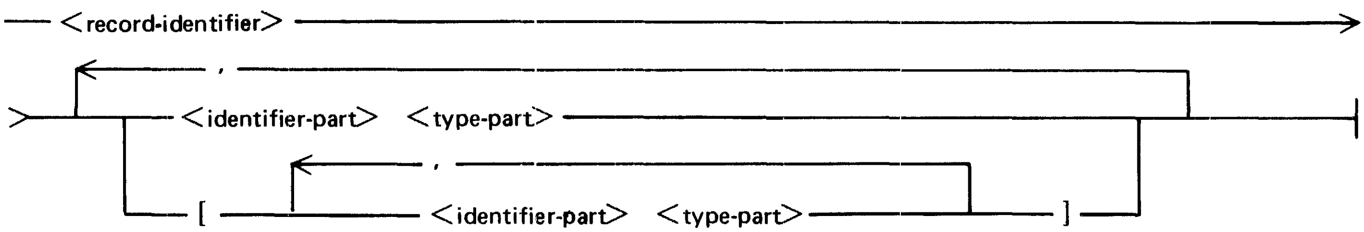
remaps-part
Refer to remaps-part in this section.

type-part
Refer to type-part in this section.

unstructured-part

The syntax and semantics of the unstructured-part in the RECORD statement are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

record-identifier
This field can be any valid SDL/UPL identifier and specifies the name of the record structure.

identifier-part
Refer to identifier-part in this section.

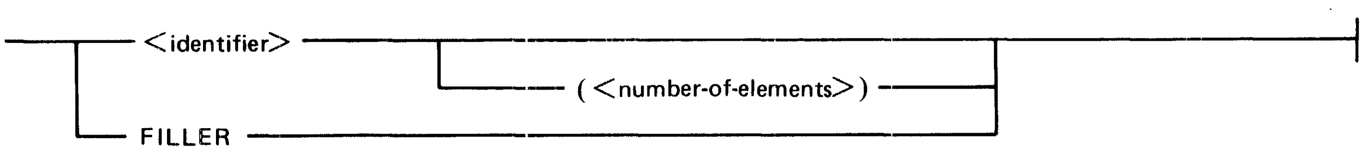
type-part
Refer to type-part in this section.

[ ]
The left and right broken bracket characters cause the enclosed identifiers to become an alternative format for the same area as that represented by the identifier specified immediately before the left and right broken bracket characters.

identifier-part

The syntax and semantics of the identifier-part in the RECORD statement are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

identifier
This field can be any valid SDL/UPL identifier and specifies the name of the data item or array.

number-of-elements

This field specifies the number of elements in the array. It can be any valid SDL/UPL number, identifier, or expression that returns a binary value.

An SDL/UPL array is a vector which is a group of memory locations associated with a single identifier. All elements of an array are identical in structure. Individual array elements are referenced by using a subscript with the array identifier.

Any identifier followed by a number in parentheses names an array.

Array subscripts are zero-relative. For example, the first element of array ARRAY is ARRAY(0). Valid subscripts for a 5-element array are 0, 1, 2, 3, and 4. If the subscript is not between 0 and n-1 inclusive, where n is the declared number of elements in the array, an invalid subscript error is generated and the program is terminated by the MCP.

The maximum number of elements per array is 65,535. Each element has a maximum length of 65,535 bits (8191 characters).

Identifiers specified as an array in the structured part of a record declaration cannot have nested record structures.

FILLER and parent field

The keyword FILLER designates the memory areas which the program does not reference. A parent identifier of an item is the field which the item subdivides. The keyword FILLER can be used on any level, specified by <level-number>, which is greater than 01. If the FILLER keyword is the last data item in a structure and its parent field specifies a length, the FILLER keyword can be omitted. The SDL/UPL compiler supplies an implied FILLER. A parent identifier of an item is the field which the item subdivides. The parent identifier must have a lower level number than its subdividing item.

remaps-part

The syntax and semantics of the remaps-part in the RECORD statement are described as follows:

SDL and UPL Syntax:

```
——— <remap-identifier>  REMAPS ——————————— <identifier> ———————————————|
```

Syntax Semantics:

remap-identifier

This field can be any valid SDL/UPL identifier and specifies the alternative name of the same memory space as <identifier>.

REMAPS

The keyword REMAPS causes the starting address of <remap-identifier> to be the same as <identifier>.

<remap-identifier> cannot be larger than <identifier>. However, it can be remapped by a smaller identifier. In that case, the SDL/UPL compiler provides implied-filler bits on the unmapped rightmost bits. There is no actual limit to the number of times a field can be remapped. <remap-identifier> can be remapped by another <remap-identifier>.

identifier
This field can be any valid SDL/UPL identifier and specifies the name of the field to be remapped.

type-part

The syntax and semantics of the type-part in the RECORD statement are described as follows:

SDL and UPL Syntax:

```
───┬── FIXED ──────────────────────────────┬─────────────────────────────┤
   ├── BIT ( <bit-size> ) ─────────────────┤
   ├── CHARACTER ( <character-size> ) ─────┤
   └── <record-identifier> ────────────────┘
```

Syntax Semantics:

BIT
The keyword BIT causes <identifier> to have a data type equal to BIT. A bit can have a value equal to 0 or 1 and is the smallest unit of data that can be addressed on the B 1000 computer system.

CHARACTER
The keyword CHARACTER causes <identifier> to have a data type equal to CHARACTER. A character is 8 bits long and represents one of the 256 EBCDIC characters.

FIXED
The keyword FIXED causes <identifier> to have a data type equal to FIXED. An identifier with a FIXED data type is 24 bits long, with the sign in the leftmost bit position, and is used for arithmetic calculations. A negative number is stored as the two's complement of its like positive number. Fixed identifiers can range from −8,388,608 to +8,388,607, inclusive.

bit-size
This field specifies the number of bits in <identifier> and can be any valid SDL/UPL number, identifier, or expression that returns a binary value.

character-size
This field specifies the number of characters in <identifier> and can be any valid SDL/UPL number, identifier, or expression that returns a binary value.

record-identifier
This field specifies the name of a record structure. This field can be any valid SDL/UPL identifier.

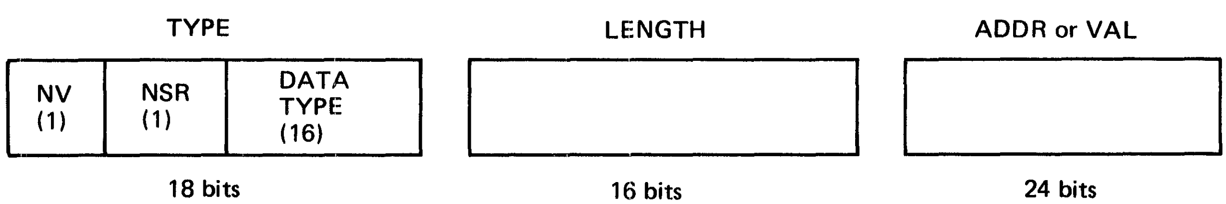
Qualified Record Names

To reference an identifier within a record, the identifier must include the name of all of its parent identifiers separated by the period (.) character.

Example:

```
RECORD TYPEFIELD
  NV         BIT(1),
  NSR        BIT(1),
  DATATYPE   BIT(16);

RECORD DESCRIPTION
  TYPE       TYPEFIELD,
  LENGTH     BIT (16),
  (ADDR       BIT(24),
  VAL        BIT(24) );

DECLARE      D       DESCRIPTION,
             A(5)    TYPEFIELD;

D := 0;
A(1) := 0;
D.TYPE.NV := @(1)1@;
A(1).NV := @(1)0@;
D.LENGTH := 4;
D.TYPE.NSR := @(1)0@;
A(4).NSR := @(1)1@;
```

In the preceding example, two record structures are specified in the DECLARE statement. They are identifier D and array A. Since identifier D and array A have no parents, each identifier is completely qualified. If field NV is to be accessed, the name must contain its parent identifiers. Because field NV has two parents, either D.TYPE.NV or A(n).NV can be specified, where n is the element number within array A. Figure 4-2 shows the data space created when identifier D is declared.



G18301

Figure 4-2. Data Space Created for Identifier D

In the record named DESCRIPTION, the previously described record named TYPEFIELD is the data type for field TYPE. This gives TYPE the subfields NV, NSR, and DATATYPE. Fields ADDR and VAL are alternative formats and, in the example, they have the same data type. The data types can vary.

Defined record identifiers can be used as data types in any DECLARE statement, including a RECORD statement.

Record-Reference Identifiers

In some cases, storage is not to be directly allocated for a record, although some program data can be in the format specified by the record structure. Record-reference identifiers provide a means to impose the record structure on a memory area during program execution.

Record-reference identifiers are bound to an identifier by the REFER verb, as simple reference identifiers are bound. Field name qualification, within a record-reference identifier, is the same as with record structure names. The record-reference identifier is the first component of a qualified name used to access a field within the record.

If the record-reference identifier is bound to an expression, the expression must generate an address.

Record-reference identifiers cannot be specified in the REDUCE verb.

The area length to which the record-reference identifier is bound must equal the length of the record structure.

Example:

```
RECORD   THIS_AND_THAT
         FIRST               FIXED,
         SECOND              BIT(3),
         THIRD               CHARACTER(10);

DECLARE  INFO THIS_AND_THAT  REFERENCE,
         BIG_AREA            BIT (800),
         X                   FIXED;

 REFER INFO TO SUBBIT(BIG_AREA,75,107);
 X := INFO.FIRST;
```

Identifier X contains a fixed-number representation of the 24 bits beginning at the 76th (bit 75) bit of the identifier BIG_AREA. Exactly 107 bits are assigned to the record-reference identifier INFO. Record identifier THIS_AND_THAT defines exactly 107 bits of information.

FILE DECLARATIONS

The FILE declaration statement describes a file to be used by a program and assigns an internal identifier to that file. More than one file attribute can be specified for each file, although all file attributes of the FILE declaration statement are optional. The default value is automatically set for any omitted file attribute.

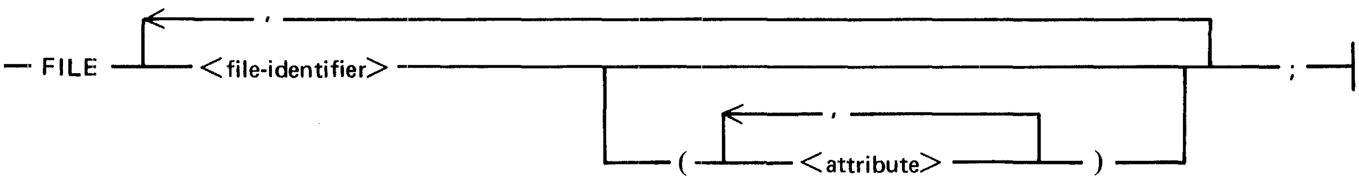
All FILE declarations must appear within the declaration portion of a program or procedure.

All underscore (_) characters used in internal file identifiers are converted to the period (.) character in the file parameter block (FPB).

A FILE declaration consists of the reserved word FILE followed by one or more file identifiers which are separated by the comma (,) character. Each file identifier is optionally followed by file attributes enclosed within parentheses "( )" characters.

The syntax and semantics of the FILE declaration are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

file-identifier
    This field can be any valid B 1000 file name and specifies the internal file name of the file.

attribute
    This field can be any valid SDL/UPL file attribute.

The valid file attributes are listed and defined in the following paragraphs.

ALL_AREAS_AT_OPEN	OPTIONAL
AREAS	PACK_ID
BUFFERS	PROTECTION
DEVICE	PROTECTION_IO
END_OF_PAGE_ACTION	RECORDS
EU_INCREMENTED	REEL
EU_SPECIAL	REMOTE_KEY
EXCEPTION_MASK	SAVE
FILE_TYPE	SECURITYTYPE
INVALID_CHARACTERS	SECURITYUSE
LABEL	SERIAL
LABEL_TYPE	TRANSLATE
LOCK	USE_INPUT_BLOCKING
MODE	USER_NAMED_BACKUP
MULTI_PACK	VARIABLE
NUMBER_OF_STATIONS	WORK_FILE
OPEN_OPTION	

ALL_AREAS_AT_OPEN

The ALL_AREAS_AT_OPEN file attribute causes the area disk space to be allocated when the file is opened. If sufficient disk space is not available, an ODT message is displayed which indicates that no more disk space is available. The program is then suspended. By default, the value of each disk area is allocated when the area is needed.

SDL and UPL Syntax:

Example:

```
FILE DISKFILE (ALL_AREAS_AT_OPEN);
```

AREAS

The AREAS file attribute assigns the number of disk areas and the number of blocks per area for a disk file.

This option applies only to disk files.

If the AREAS and RECORDS file attributes are not specified, the SDL/UPL compiler assigns a value equal to 100 for the records per area.

SDL and UPL Syntax:

```
——AREAS = <number-of-areas>/<blocks-per-area>——————————————————————————|
```

Syntax Semantics:

number-of-areas
: This field can be any number and specifies the allowed number of disk areas for the file. The default value is 25.

blocks-per-area
: This field can be any number and specifies the number of blocks each area can have. The default value is 100.

/
: The virgule (/) character is a delimiter and is not the division operator.

Example:

```
FILE DISKFILE (AREAS = 50/200);
```

BUFFERS

The BUFFERS file attribute specifies the number of input/output (I/O) buffers to be assigned to the file. The BUFFERS file attribute cannot be specified for a file with a device type equal to QUEUE.

SDL and UPL Syntax:

```
—— BUFFERS = <number-of-buffers>———————————————————————————————————————|
```

Semantics:

number-of-buffers
: This field can be any number and specifies the number of buffers for the file. The default value is 1.

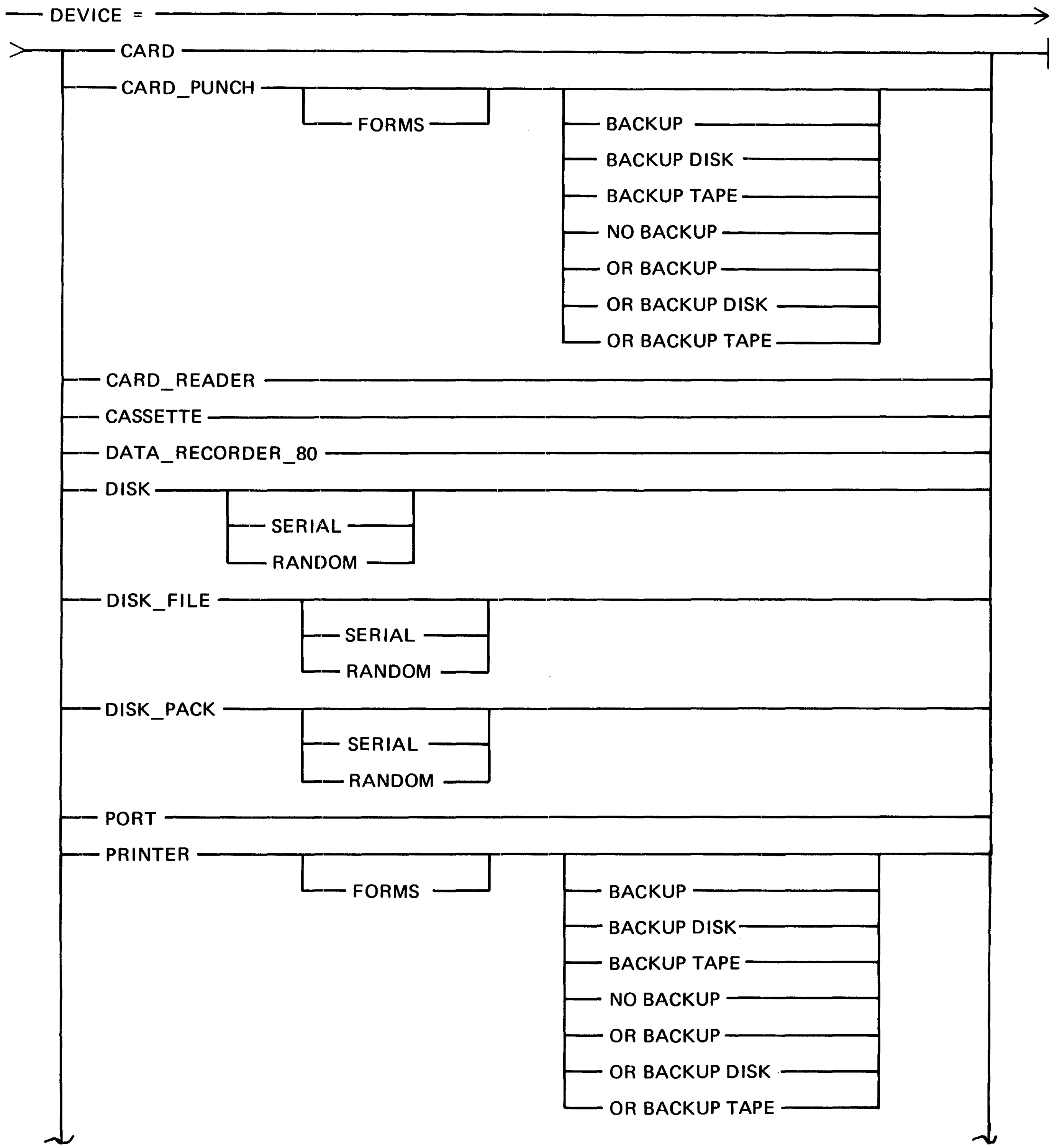
Example:

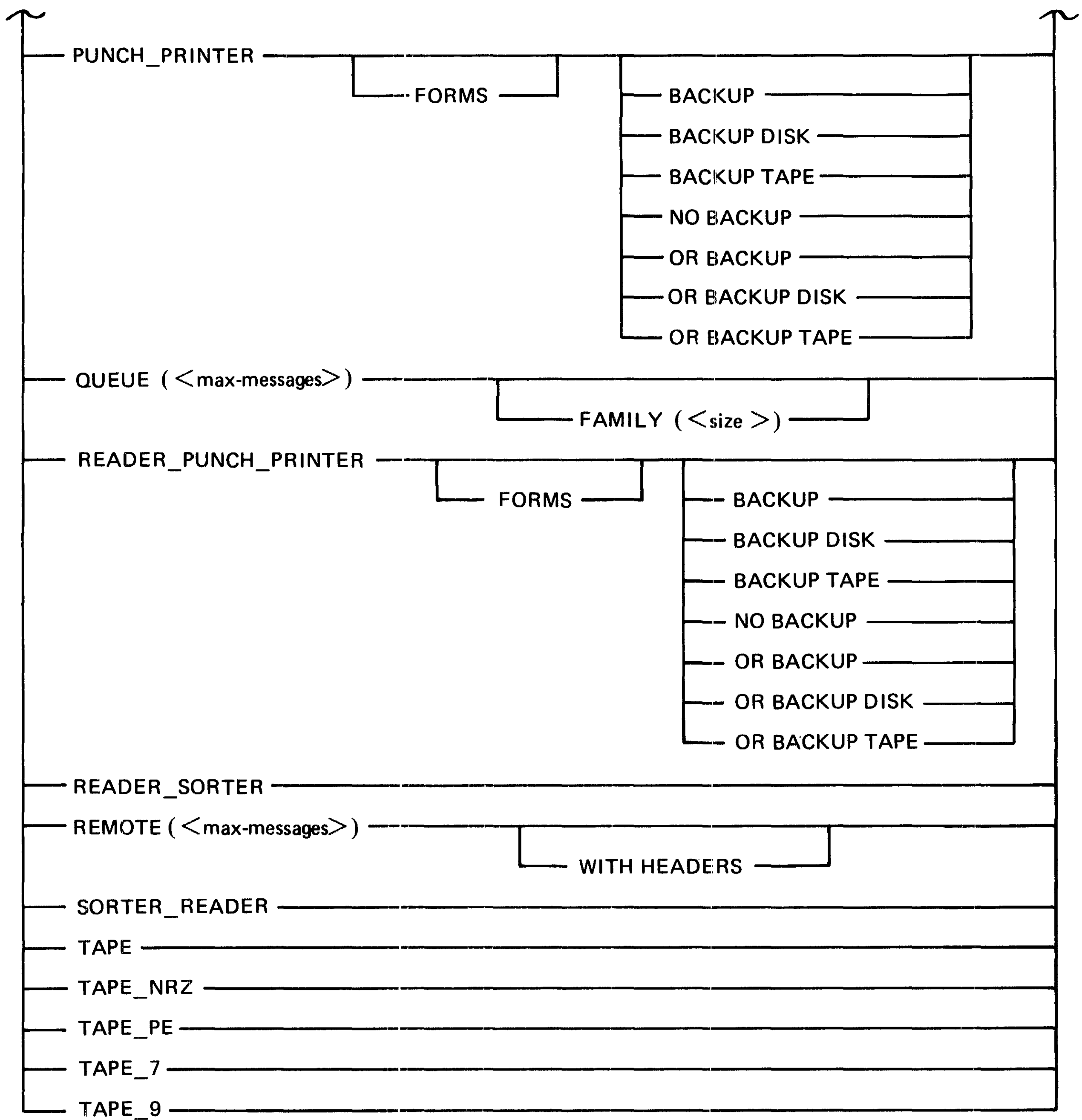
```
FILE DISKFILE (BUFFERS = 2);
```

DEVICE

The DEVICE file attribute specifies the type of input/output (I/O) device on which the file is to reside.

SDL and UPL Syntax:




PUNCH_PRINTER
FORMS
BACKUP
BACKUP DISK
BACKUP TAPE
NO BACKUP
OR BACKUP
OR BACKUP DISK
OR BACKUP TAPE
QUEUE (<max-messages>)
FAMILY (<size >)
READER_PUNCH_PRINTER
FORMS
BACKUP
BACKUP DISK
BACKUP TAPE
NO BACKUP
OR BACKUP
OR BACKUP DISK
OR BACKUP TAPE
READER_SORTER
REMOTE (<max-messages>)
WITH HEADERS
SORTER_READER
TAPE
TAPE_NRZ
TAPE_PE
TAPE_7
TAPE_9

Syntax Semantics:

BACKUP
The keyword BACKUP causes the printer or punch file to be written to the designated printer or punch backup device. The designated printer or punch backup device is set by the MCP options PBD (Printer/Punch Backup Disk) and PBT (Printer/Punch Backup Tape).

BACKUP DISK
The keywords BACKUP DISK cause the printer or punch file to be written to the backup disk device. The MCP option PBD must be set.

BACKUP TAPE
The keywords BACKUP TAPE cause the printer or punch file to be written to the backup tape device. The MCP option PBT must be set.

CARD
The keyword CARD specifies that the device type of the file is a card reader. This keyword is the same as the CARD_READER keyword.

CARD_PUNCH
The keyword CARD_PUNCH specifies that the device type of the file is a card reader and card punch.

CARD_READER
The keyword CARD_READER specifies that the device type of the file is a card reader. This keyword is the same as the CARD keyword.

CASSETTE
The keyword CASSETTE specifies that the device type of the file is a cassette.

DATA_RECORDER_80
The keyword DATA_RECORDER_80 specifies that the device type of the file is an 80-column card reader.

DISK
The keyword DISK specifies that the device type of the file is disk. This keyword is the same as the DISK_FILE keyword.

DISK_FILE
The keyword DISK_FILE specifies that the device type of the file is disk. The keyword is the same as the DISK keyword.

DISK_PACK
The keyword DISK_PACK specifies that the device type of the file is disk pack.

FAMILY
The keyword FAMILY causes a group of subqueues to be assigned to the queue file.

FORMS
The keyword FORMS specifies that the printer or punch file has a special form. Operator action must be taken to insure that the special form is on the device before writing to the file.

max-messages
This field specifies the total number of messages that can be written to the file by another program or process before the file becomes full. This field applies to files that have a device type equal to QUEUE or REMOTE.

NO BACKUP
The keywords NO BACKUP specify that the printer or punch file is not to be written to a printer or punch backup device.

OR
The keyword OR specifies that additional backup keywords follow. These keywords are BACKUP, BACKUP DISK, BACKUP TAPE and NO BACKUP.

PORT
The keyword PORT specifies that the device type of the file is a BNA port file.

PRINTER
The keyword PRINTER specifies that the device type of the file is a line printer.

PUNCH_PRINTER
The keyword PUNCH_PRINTER specifies that the device type of the file is a card punch and card interpreter.

QUEUE
The keyword QUEUE specifies that the device type of the file is a queue.

A queue file is a temporary file structure maintained as an input and output file. Queue files are accessed with read and write operations that are conceptually identical to I/O operations which are performed on all other devices. Queue files can be declared as a family of files.

A queue file is a specialized file structure maintained by the MCP as a means of Inter-Process Communication (IPC). A queue file contains a list of messages (possibly an empty list) to which messages can be written and from which messages can be read. Queue files have a head and a tail record. The head (top) of a queue file is the first message in a queue. This is the message that is accessed by a read operation and generally is the message that has been in the queue file the longest time. The tail (end) of a queue file is the last message in the queue file to which the next written message is linked. A queue file is basically a first-in, first-out (FIFO) structure.

Queue files can be shared by several programs. When a queue file is opened, the queue driver in the MCP compares the 20-character file identifier with the names of already opened queue files. If the named queue file is opened by another program or process, the queue file is linked to the existing queue file and the USER_COUNT field in the disk file header is incremented. If the queue file is not opened, a new queue file is created as described by the parameters in the file parameter block (FPB). When a queue file is shared by several programs, the program that originally opened the queue file controls all file attributes of that queue file.

Messages stored in a queue file can reside on disk or in memory. At the time the queue file is created, an area of system disk is obtained for the queue. This area is of sufficient size to hold the entire queue. Queue file messages are stored on disk if one of the following situations occurs.

- The message being written to the queue file makes the count of messages in the queue file greater than the number of buffers for the queue file. In this case, the tailmost message is written to disk.

- The B 1000 memory management system needs the space used by an infrequently-accessed queue file. Therefore, it rolls the messages out to disk.
  Messages are stored in a variable-length format. Any record whose length is less than the declared record-size uses only the amount of memory required to write the message.

  Messages are stored in a queue file as a linked list of message descriptors. Each message descriptor is an 80-bit system descriptor with two additional link fields. The system descriptor describes the text of the message according to standard MCP conventions.

  When a queued message is in S-memory, it is stored in a memory link called a message buffer. No queue file can have more than the declared number of messages in the buffer, including messages that are being moved between disk and S-memory. The buffers are allocated from a common pool of empty buffers.

READER_PUNCH_PRINTER
The keyword READER_PUNCH_PRINTER specifies that the device type of the file is a card reader, card punch, and card interpreter.

READER_SORTER
The keyword READER_SORTER specifies that the device type of the file is a reader sorter.

REMOTE
The keyword REMOTE specifies that the device type of the file is remote. Files that have a device type equal to REMOTE can read and write messages to the network controller.

Examples:

```
FILE ANNOD (DEVICE = REMOTE);
READ ANNOD (Message);

FILE ANNOD (DEVICE = REMOTE(20) WITH HEADERS);
READ ANNOD (Buffer);
Message := SUBSTR(Buffer,49);

FILE  ANNOD (DEVICE = REMOTE(20), REMOTE_KEY,
      NUMBER_OF_STATIONS = 2);
WRITE ANNOD [C020C7000] ("message");
```

size
This field can be any valid number and specifies the number of subqueues or queue-file families in the file with a device type equal to QUEUE.

Queue-file families are a group of queues that share I/O descriptors. A group of queues have a multi-file-identifier and are accessed as a subfield of the queue-file family. A subscript must be specified in order to identify the subqueue in a queue-file family for read or write operations.

Queue-file families are declared with the FAMILY keyword.

Each member of the queue-file family is accessed with a numeric key, based on the order in which the queues are declared. The first subqueue has number 0 and the last has number n-1, where n is the number of subqueues. Specifying an index of −1 requests an unspecified read from the queue-file family. An unspecified read operation scans through the queues and returns the top message from the first non-empty queue in the family.

SORTER_READER
The keyword SORTER_READER specifies that the device type of the file is a reader sorter.

TAPE
The keyword TAPE specifies that the device type is tape.

TAPE_NRZ
The keyword TAPE_NRZ specifies that the device type is tape with the Non-Return to Zero (NRZ) recording mode.

TAPE_PE
The keyword TAPE_PE specifies that the device type of the file is tape with the phase encoded (PE) recording mode.

TAPE_7
The keyword TAPE_7 specifies that the device type of the file is a 7-channel tape.

TAPE_9
The keyword TAPE_9 specifies that the device type of the file is a 9-channel tape.

WITH HEADERS
The keywords WITH HEADERS applies only to remote files and specifies that a 50-byte message header is supplied/expected in all read and write operations to the remote file.

Examples:

```
FILE OUT_MASTER (DEVICE = PRINTER       % The file OUT_MASTER is
                 OR BACKUP DISK         % printed if the line printer
                 OR BACKUP TAPE);       % is available. Otherwise, a
                                        % backup output file is
                                        % created on disk or tape.

FILE SUMMARY (LABEL = "PAYROLL"/"W2",   % The two files, W_2_SUMMARY
              DEVICE = DISK_PACK),      % and W_2_REPORT, are declared.
     REPORT (DEVICE = PRINTER FORMS     % W_2_SUMMARY has the label
             OR BACKUP DISK);           % PAYROLL/W2 and device type of
                                        % DISK_PACK.  W_2_REPORT has
                                        % the device type of PRINTER
                                        % and special forms with the
                                        % BACKUP DISK option.
```

END_OF_PAGE_ACTION

The END_OF_PAGE_ACTION file attribute causes the write operations to return the end-of-file exception when the end of page is encountered on the line printer. The program can specify action to be taken with ON EOF keywords in the WRITE verb. The default is no automatic end-of-page reporting.

SDL and UPL Syntax:

```
—— END_OF_PAGE_ACTION ——————————————————————————————|
```

Example:

```
FILE DISKFILE (DEVICE = DISK,
               EU_INCREMENTED = 2);
```

EU_INCREMENTED

The EU_INCREMENTED file attribute specifies the disk drive on which the first area of a file is to be written. Each subsequent area is then written on the next drive. If the next drive does not exist, the next area of the file is written to the first drive and so on. By default, files are written to one disk drive.

SDL and UPL Syntax:

```
—— EU_INCREMENTED = <drive-number> ——————————————————|
```

Syntax Semantics:

drive-number
: This field can be any valid number within the range 0 to 15 and specifies the disk drive number of a head-per-track or systems disk pack. If <drive-number> is not an available disk pack, then 0 is used.

Example:

```
FILE LINE (DEVICE = PRINTER,
           END_OF_PAGE_ACTION);
```

EU_SPECIAL

The EU_SPECIAL file attribute specifies the disk drive on which the file is to be written. By default, areas of the file are allocated anywhere on disk.

SDL and UPL Syntax:

```
—— EU_SPECIAL = <drive-number> ——————————————————————|
```

Syntax Semantics:

drive-number
: This field can be any number within the range 0 to 15 and specifies the disk drive on which the file is to be written. Only head-per-track and systems disk packs are valid. If the drive is not available, <drive-number> is set to 0.

Example:

```
FILE DISKFILE (DEVICE = DISK,
               EU_SPECIAL = 2);
```

EXCEPTION_MASK

The EXCEPTION_MASK file attribute specifies the types of exceptions that the program can handle for the file. By default, no exceptions are to be reported in the exception mask.

SDL and UPL Syntax:

```
── EXCEPTION_MASK = <exception-bits> ──────────────────────────┤
```

Syntax Semantics:

exception-bits
  This field must be a 24-bit value. Each bit signifies which exception is to be reported in the exception mask field for read and write operations. The default value is @000000@.

Example:

```
FILE DISKFILE (DEVICE =DISK,
               EXCEPTION_MASK = @FFF000@);
```

FILE_TYPE

The FILE_TYPE file attribute specifies the file type of the created file. In particular, B 1000 compilers specify a FILE_TYPE = CODE for their resulting code files. The default is DATA.

SDL and UPL Syntax:

```
── FILE_TYPE = ──┬── DATA ──────────┬──────────────────────────┤
                 ├── INTERPRETER ───┤
                 ├── CODE ──────────┤
                 ├── INTRINSIC ─────┤
                 └── PSR_DECK ──────┘
```

Syntax Semantics:

CODE
  The keyword CODE causes the file being created to be a code file.

DATA
  The keyword DATA causes the file being created to be a data file.

INTERPRETER
  The keyword INTERPRETER causes the file being created to be an interpreter file.

INTRINSIC
  The keyword INTRINSIC causes the file being created to be an intrinsic file.

PSR__DECK
The keyword PSR__DECK causes the file being created to be a pseudo-reader file.

Example:

```
HOST_NAME = "HOSTA"
```

HOST__NAME

The HOST__NAME file attribute specifies that the file resides on a remote BNA host system.

SDL and UPL Syntax:

```
— HOST_NAME = "<host-name>" ————————————————————————————————|
```

Syntax Semantics:

host-name
The field can be any character string up to 17 characters long which specifies the name of the remote host system.

Example:

```
FILE OUT (DEVICE = DISK,
          FILE_TYPE = CODE);
```

INVALID__CHARACTERS

The INVALID__CHARACTERS file attribute applies only to printer files and specifies the type of invalid-character reporting that is to be done.

When a printer file includes a print character that is not valid on the line printer, an invalid-character exception is reported to the MCP. The value of the INVALID__CHARACTERS file attribute determines the action taken when invalid characters are encountered while printing a file. The default value is 2.

SDL and UPL Syntax:

```
— INVALID_CHARACTERS = ——┬—— 0 ——┬——————————————————————————|
                         ├—— 1 ——┤
                         ├—— 2 ——┤
                         └—— 3 ——┘
```

Syntax Semantics:

0
: The keynumber 0 causes the MCP to report all printed lines containing invalid characters.

1
: The keynumber 1 causes the MCP to report the first print line containing any invalid characters and then to terminate the program.

2
: The keynumber 2 causes the MCP to report only the first print line that contains any invalid characters and to continue printing.

3
: The keynumber 3 causes the MCP not to report any print lines that contain invalid characters.

Example:

```
FILE LINE (DEVICE = PRINTER,
           INVALID_CHARACTERS = 3);
```

LABEL

The LABEL file attribute specifies an external file name for the file as it appears, or as it is to be stored in the disk directory. The file identifier in the FILE declaration statement is the default name. The LABEL file attribute writes the file identifiers in the file parameter block (FPB).

If only the multi-file-identifier is specified, the file identifier is assigned blank characters.

The pack identifier is not affected by the LABEL file attribute.

The MCP uses only the first ten characters of each identifier.

SDL and UPL Syntax:

```
— LABEL = "<multi-file-identifier>" ———————————————————————————————————|
                                       |                            |
                                       └—— / "<file-identifier>" ———┘
```

Syntax Semantics:

multi-file-identifier
: This field can be any valid 10-character identifier that follows the B 1000 file-naming convention.

file-identifier
: This field can be any valid 10-character identifier that follows the B 1000 file-naming convention.

Example:

```
FILE DISKFILE (DEVICE = DISK,
               LABEL = "MASTER"/"FILE");
```

LABEL_TYPE

The LABEL_TYPE file attribute is valid only for tape and printer files and specifies the label type of the file. The BURROUGHS standard label and the ANSI standard label are the same. The default LABEL_TYPE label is the ANSI standard label.

SDL and UPL Syntax:

```
— LABEL_TYPE = ——┬—— UNLABELED ——┬——————————————————————————|
                 ├—— BURROUGHS ——┤
                 └—— ANSII ——————┘
```

Syntax Semantics:

UNLABELED
    The keyword UNLABELED causes the file to be unlabeled.

BURROUGHS
    The keyword BURROUGHS causes the file to have the Burroughs standard label.

ANSI
    The keyword ANSI causes the file to have the ANSI standard label.

Example:

```
FILE LINE (DEVICE = PRINTER,
           LABEL_TYPE = BURROUGHS);
```

LOCK

The LOCK file attribute requests the MCP to enter the external file name into the disk directory. The LOCK file attribute is overridden if the file is closed with the purge option.

There are two ways to permanently close a file: with the CLOSE verb, or with an implied close when the program goes to end of job.

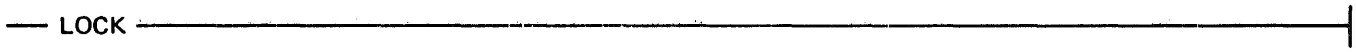
If a tape or disk file is explicitly closed and the LOCK file attribute is specified in the file declaration, the file identifier remains in the disk directory. The LOCK file attribute is used to close the file when either a CLOSE REMOVE; or CLOSE CRUNCH; statement is specified. The LOCK file attribute is not used to close the file when CLOSE PURGE; statement is specified.

An implied close occurs under two conditions: when a program goes to end of job with the file still open and when a program is discontinued by using the MCP commands DS or DP. A file is not closed if the system halts.

If an implied close occurs, the file is locked into the disk directory only if the LOCK file attribute is specified. If not, the file is closed with the release option. Only new files are not entered in the disk directory if the LOCK file attribute is not specified and the file is implicitly closed.

The default is no LOCK.

SDL and UPL Syntax:



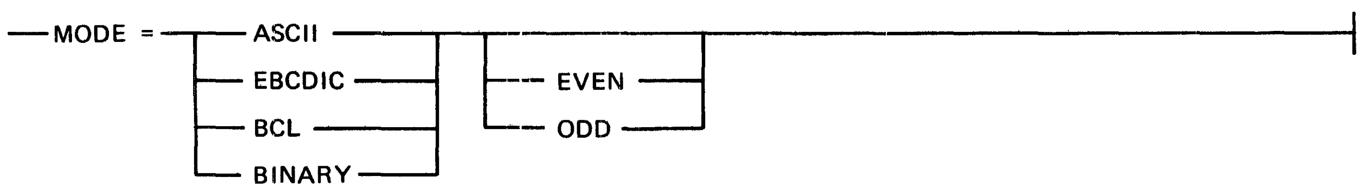
Example:

```
FILE DISKFILE (DEVICE = CISK,
               LOCK);
```

MODE

The MODE file attribute specifies the type of parity checking and translation that is to be used for the file. The default is odd parity checking or EBCDIC translation, whichever is applicable.

SDL and UPL Syntax:



Syntax Semantics:

ODD
  The keyword ODD specifies that odd-parity checking is to be used.

EVEN
  The keyword EVEN specifies that even-parity checking is to be used.

EBCDIC
  The keyword EBCDIC specifies that EBCDIC translation is to be used.

ASCII
  The keyword ASCII specifies that ASCII translation is to be used.

BCL
  The keyword BCL specifies that BCL translation is to be used.

BINARY
  The keyword BINARY specifies that BINARY translation is to be used.

Example:

```
FILE TAPEFILE (DEVICE = TAPE,
               MOCE = CCC);
```

MULTI_PACK

The MULTI_PACK file attribute specifies that a single file can reside on more than one disk pack. The default is that the entire file must reside on one disk pack.

SDL and UPL Syntax:

```
—— MULTI_PACK ————————————————————————————————————————|
```

Example:

```
FILE DISKFILE (DEVICE = DISK,
               MULTI_PACK);
```

NUMBER_OF_STATIONS

The NUMBER_OF_STATIONS file attribute specifies the maximum number of stations that are attached to this remote file. The maximum number of stations that can be attached is system dependent and is determined by the network controller. The NUMBER_OF_STATIONS file attribute must not specify more stations than the network controller has defined. The default is 1.

SDL and UPL Syntax:

```
—— NUMBER_OF_STATIONS = <number> ——————————————————————|
```

Syntax Semantics:

number
This field specifies the maximum number of stations that are to be attached to the remote file when the remote file open is approved by the network controller.

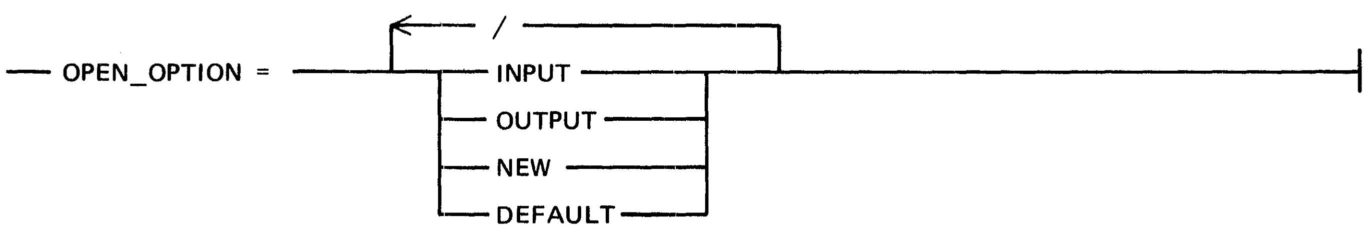
Example:

```
FILE REMOTEFILE (DEVICE = REMOTE,
                 NUMBER_OF_STATIONS = 5);
```

OPEN_OPTION

The OPEN_OPTION file attribute specifies how the file is to be opened.

SDL and UPL Syntax:



Syntax Semantics:

INPUT
The keyword INPUT causes the file to be opened input.

OUTPUT
The keyword OUTPUT causes the file to be opened output.

NEW
The keyword NEW causes the file to be opened as a new file.

DEFAULT
The keyword DEFAULT causes the file to be opened using the following default options for each device.

Device	**Option**
CARD	INPUT
PRINTER	OUTPUT
PUNCH	OUTPUT
DISK	INPUT
REMOTE	INPUT/OUTPUT
TAPE	INPUT
QUEUE	INPUT/OUTPUT

Example:

```
FILE DISKFILE (DEVICE = DISK,
               OPEN_OPTION = INPUT/OUTPUT/NEW);
```

OPTIONAL

The OPTIONAL file attribute specifies that the file can be missing without suspending program execution.

Performing a read operation from a missing file generates the ODT message FILE MISSING. If the OPTIONAL file attribute is specified, the MCP command OF (optional file) causes the program to perform the ON EOF branch for any read of the file. Program execution then continues. The default is no OPTIONAL which requires the file to be present.

SDL and UPL Syntax:

```
— OPTIONAL ——————————————————————————————————|
```

Example:

```
FILE DISKFILE (DEVICE = DISK,
               OPTIONAL);
```

PACK_ID

The PACK_ID file attribute specifies the disk-pack identifier for the disk file. The default pack identifier is the system disk.

SDL and UPL Syntax:

── PACK_ID = "<pack-identifier>" ──────────────────────┤

Syntax Semantics:

pack-identifier
    This field can be any identifier that follows the B 1000 disk file naming convention for disk files.

Example:

```
FILE DISKFILE (DEVICE = DISK,
               PACK_ID = "USER");
```

PROTECTION

The PROTECTION file attribute specifies a security type to the file. The default is 0.

SDL and UPL Syntax:

── PROTECTION = <number> ──────────────────────┤

Syntax Semantics:

number
    This field can be any number between 0 and 4, inclusive, and is used to define the security type. The security type for each value is listed in the following table.

Value	**Security Type**
0	Default
1	Public
2	Private
3	Guard

Example:

```
FILE DISKFILE (DEVICE = DISK,
               PROTECTION = 2);
```

PROTECTION_IO

The PROTECTION_IO file attribute specifies whether the file is to be opened input, output, or both.

SDL and UPL Syntax:

```
── PROTECTION_IO = <number> ──────────────────────────────────────┤
```

Syntax Semantics:

number
This field can be any number between 0 and 2, inclusive. The meaning of each value of <number> follows.

Value	Definition
0	Input/Output (Default)
1	Input
2	Output

Example:

```
FILE DISKFILE (DEVICE = DISK,
               PROTECTION_IO = 2);
```

RECORDS

The RECORDS file attribute specifies the number of characters per record or per block.

The default values in bytes for each device follow.

Device	Bytes
CARD	80
DISK	180
PRINTER	132
ODT	72
All Others	80

SDL and UPL Syntax:

```
── RECORDS = ──┬── <physical-size> ─────────────────────────┬──────┤
               └── <logical-size>/<records-per-block> ──────┘
```

Syntax Semantics:

physical-size
This field can be any number and specifies the number of characters per block.

logical-size
This field can be any number and specifies the number of characters per record.

records-per-block
This field can be any number and specifies the number of records per block.

Example:

```
FILE DISKFILE (DEVICE = DISK,
              RECORDS = 180/10);
```

REEL

The REEL file attribute applies only to magnetic tape files and specifies the starting reel number.

For output tape files, the MCP uses the supplied reel number as the starting reel number. This reel number is written in the tape label. If more than one physical tape is needed to hold the file, the MCP automatically increments the reel number by one and writes the new reel number in the label of the next tape.

For input tape files, the MCP starts reading the tape file at the specified reel number. This means that the MCP looks for the tape whose label contains the same reel number as that specified in the REEL file attribute, as well as the name of the requested file. As in output, the MCP automatically increments the reel number by one if the physical tape has been read but the actual end of file has not been reached.

The default reel number is 1.

SDL and UPL Syntax:

— REEL = <reel-number> ——————————————————————|

Syntax Semantics:

reel-number
This field can be any number and specifies the starting reel number in which to read or write.

Example:

```
FILE TAPEFILE (DEVICE = TAPE,
              REEL = 5);
```

REMOTE_KEY

The REMOTE_KEY file attribute directs read and write operations to specific stations. The NUMBER_OF_STATIONS file attribute must be specified in conjunction with the REMOTE_KEY file attribute. The remote key is a 10-character field containing station number, message length, and message type. This 10-character field is the <remote-key-identifier> field in the syntax for the READ and WRITE verbs. The following is the format of the remote key.

Remote Key Fields	Data Type	Length in Bytes	Value Range
Station number	CHARACTER	3	1 – 999
Message length (bytes)	CHARACTER	4	0 – 4095
Message type	CHARACTER	3	000 (write) or 001 (read)

The default is no REMOTE_KEY.

SDL and UPL Syntax:

```
—— REMOTE_KEY ————————————————————————————————————————|
```

Example:

```
FILE REMOTEFILE (DEVICE = REMOTE,
                 REMOTE_KEY,
                 NUMBER_OF_STATIONS = 4);
```

SAVE

The SAVE file attribute specifies the number of days the declared file is to be saved. Files are never removed from the system automatically. The default is 30.

SDL and UPL Syntax:

```
—— SAVE = <number-of-days> ————————————————————————————|
```

Syntax Semantics:

number-of-days
    This field can be any number and specifies the number of days to save the disk file.

Example:

```
FILE DISKFILE (DEVICE = DISK,
               SAVE = 45);
```

SECURITYTYPE

The SECURITYTYPE file attribute specifies a security type to the file. The default is 0.

SDL and UPL Syntax:

```
—— SECURITYTYPE = <number> ——————————————————————————————————————|
```

Syntax Semantics:

number
This field can be any number between 0 and 4, inclusive, and is used to define the security type. The security type for each value is listed in the following table.

Value	Security Type
0	Default
1	Public
2	Private
3	Guard

Example:

```
FILE DISKFILE (DEVICE = DISK,
               SECURITYTYPE = 2);
```

SECURITYUSE

The SECURITYUSE file attribute specifies whether the file is to be opened input, output, or both.

SDL and UPL Syntax:

```
—— SECURITYUSE = <number> ———————————————————————————————————————|
```

Syntax Semantics:

number
This field can be any number between 0 and 2, inclusive. The meaning of each value of <number> follows.

Value	Definition
0	Input/Output (Default)
1	Input
2	Output

Example:

```
FILE DISKFILE (DEVICE = DISK,
               SECURITYUSE = 2);
```

SERIAL

The SERIAL file attribute specifies the serial number of the output media. This media can be tape or disk. The default is no serial number.

SDL and UPL Syntax:

```
── SERIAL = ──┬── <number> ──────────────────┬──────────────────────┤
              └── "<character-string>" ──────┘
```

Syntax Semantics:

number
: This field can be any valid number and specifies the serial number for the output media.

character-string
: This field can be any character string and specifies the serial number for the output media.

Examples:

```
FILE TAPEFILE (DEVICE = TAPE,
               SERIAL = 123456);

FILE TAPEOUT (DEVICE = TAPE,
              SERIAL = "OUTPUT");
```

TRANSLATE

The TRANSLATE file attribute specifies that a translation is to be performed on the file by the MCP.

The MCP supplies a multi-file-identifier to the specified file identifier. The multi-file-identifier is TRANSLATE.

The TRANSLATE file attribute sets the translate boolean in the file parameter block (FPB).

SDL and UPL Syntax:

```
── TRANSLATE = "<file-identifier>" ─────────────────────────────────┤
```

Syntax Semantics:

file-identifier
: This field can be any valid file identifier that follows the B 1000 file naming convention and specifies the name of the file that contains the translate table.

Example:

```
FILE TFILE (DEVICE = DISK,                  % The resulting translate
            TRANSLATE = "TRANSFILE");       % file identifier is
                                            % TRANSLATE/TRANSFILE
```

USE_INPUT_BLOCKING

The USE_INPUT_BLOCKING file attribute applies only to input disk, tape, or card files.

For disk files, the record and block size specifications are taken from the disk file header (DFH). Any specifications for these file attributes are ignored.

For tape files, the record and block size specifications are taken from the tape label. If this option is used for an unlabeled tape file, a run-time error results.

For card files, the following record lengths are assumed.

Number of Columns	Length
80	80 Bytes
96	96 Bytes
BIN	960 Bits

The default is the record and block sizes that are specified in the file declaration. Those options omitted are set to default values.

SDL and UPL Syntax:

```
—— USE_INPUT_BLOCKING ——————————————————————————————————|
```

Example:

```
FILE DISKFILE (DEVICE = DISK,
               USE_INPUT_BLOCKING);
```

USER_NAMED_BACKUP

The USER_NAMED_BACKUP file attribute specifies that if the printer file goes to backup, the name of the printer backup file is the name specified by the LABEL file attribute; otherwise a system backup number generated by the system. The default uses the system-assigned backup file names.

SDL and UPL Syntax:

```
—— USER_NAMED_BACKUP ——————————————————————————————————|
```

Example:

```
FILE LINE (DEVICE = PRINTER BACKUP DISK,
           USER_NAMED_BACKUP,
           LABEL = "LINE"/"BACKUP");
```

VARIABLE

The VARIABLE file attribute specifies that the file has variable-length records. The default is fixed-length records.

SDL and UPL Syntax:

```
—— VARIABLE ——————————————————————————————————————————|
```

Example:

```
FILE DISKFILE (DEVICE = DISK,
               VARIABLE);
```

WORK_FILE

The WORK_FILE file attribute causes the job number of the program to be included as part of the file identifier. Workfiles are temporary files associated with a specific job and are removed when the program goes to end of job. The default is no workfile.

SDL and UPL Syntax:

```
—— WORK_FILE ——————————————————————————————————————————|
```

Example:

```
FILE DISKFILE (DEVICE = DISK,
               WORK_FILE);
```

SWITCH_FILE DECLARATION

The switch-file declaration statement groups files together under a single file identifier. All files grouped into a switch file must be declared in a file declaration statement before they can be referenced in the switch-file declaration statement.

A subscripted switch-file identifier is valid anywhere a file identifier is valid.

If there are n files in a switch-file group, the subscript must range from 0 to n-1. The subscript selects a file from the switch-file group, based on physical order. The first file in the list (from the left) is switch file zero and the last is switch file n-1.

If all the files in a switch-file group are declared with a device type equal to REMOTE, then the REMOTE_KEY file attribute can be used with the switch-file identifier. If all the files in the switch-file group are not declared with a device type equal to REMOTE, then the REMOTE_KEY file attribute cannot be used.

SDL and UPL Syntax:

```
                                                  ┌←———— , ————————————┐
—— SWITCH_FILE <switch-file-identifier> ( ————————┴——— <file-identifier> ——┴—— ); ——|
```

Syntax Semantics:

switch-file-identifier

This field can be any valid SDL/UPL file identifier and specifies the name of the switch file.

file-identifier

This field can be any valid SDL/UPL file identifier and specifies the name of the file that is to belong to the group of files in the switch file. Example Program:

Example Program:

```
FILE CARDS (DEVICE = CARD),
     TAPEI (DEVICE = TAPE,
            USE_INPUT_BLOCKING),
     DISKI (DEVICE = DISK,
            USE_INPUT_BLOCKING),
     PUNCH (DEVICE = PUNCH),
     LINE  (DEVICE = PRINTER),
     TAPEO (DEVICE = TAPE,
            RECORDS = 80/4),
     DISKO (DEVICE = DISK,
            RECORDS = 80/9);

SWITCH_FILE INPUT (CARDS, TAPEI, DISKI),
            OUTPUT (PUNCH, LINE, TAPEO, DISKO);

DECLARE INPUT_TYPE  FIXED,
        OUTPUT_TYPE FIXED,
        ODT_INPUT   CHARACTER (3),
        BUFFER      CHARACTER (80);

DISPLAY ("ENTER INPUT TYPE OR ENTER BYE TO GO TO END OF JOB");
ACCEPT ODT_INPUT;
IF ODT_INPUT = "BYE" THEN DO;
                              DISPLAY ("GOOD BYE");
                              STOP;
                          END;
INPUT_TYPE :=  BINARY (SUBSTR (ODT_INPUT, 0, 1)) MOD 3;
DISPLAY ("ENTER OUTPUT TYPE OR ENTER BYE TO GO TO END OF JOB");
ACCEPT ODT_INPUT;
IF ODT_INPUT = "BYE" THEN DO;
                              DISPLAY ("GOOD BYE");
                              STOP;
                          END;
OUTPUT_TYPE :=  BINARY (SUBSTR (ODT_INPUT, 0, 1)) MOD 3;
OPEN INPUT (INPUT_TYPE) INPUT;
OPEN OUTPUT (OUTPUT_TYPE) WITH OUTPUT, NEW;
DO FOREVER;
   READ INPUT (INPUT_TYPE) (BUFFER);
        ON EOF UNDO;
   WRITE OUTPUT (OUTPUT_TYPE) (BUFFER);
END;
CLOSE OUTPUT (OUTPUT_TYPE) WITH LOCK;
STOP;
FINI;
```

SECTION 5
DEFINES

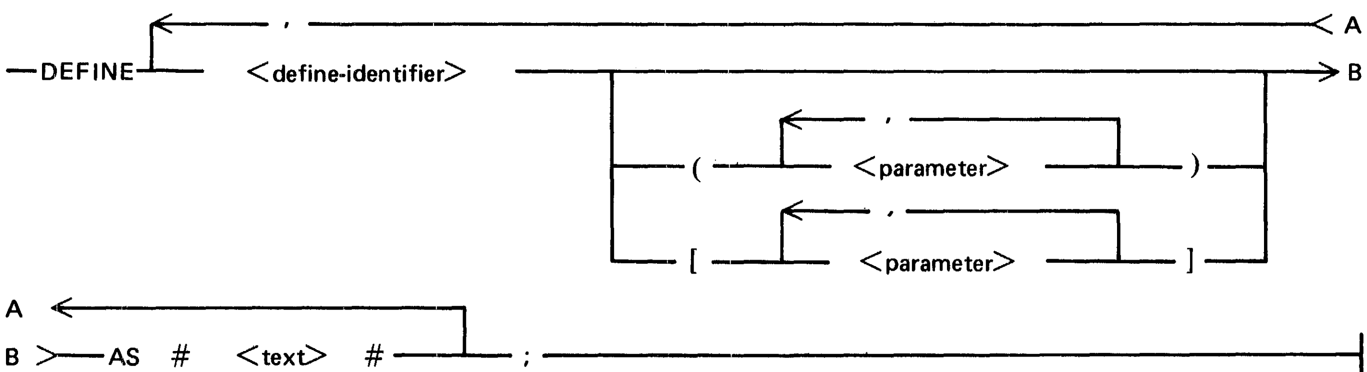
The define statement provides SDL/UPL programs with a macro definition facility by assigning a portion of the SDL/UPL source statements to an identifier.

At compile time, every occurrence of define-identifier is textually replaced by a portion of the source statement specified in <text>. If the compiler control option DETAIL is set, these macro expansions are included in the source listing. If the DETAIL option is not set, only <define-identifier> is listed. If the compiler control options EXPAND_DEFINES and XREF are set, the macro expansions are cross referenced.

The SDL/UPL compiler does not check the syntax of the <text> contents. When <define-identifier> is invoked, <text> must conform to the syntactical requirements of the statement containing <define-identifier>.

<define-identifier> can be nested within another DEFINE statement. Twelve levels of nesting are allowed.

SDL and UPL Syntax:



Syntax Semantics:

define-identifier

This field can be any valid SDL/UPL identifier and specifies the definition identifier. Reserved words cannot be specified as <define-identifier>. However, <define-identifier> can be defined as a reserved word. Special words can be redefined and only lose their special significance within the scope of the definition. Refer to Appendix A for a complete list of reserved and special words recognized by the SDL/UPL compiler.

parameter

This field can be any valid SDL/UPL identifier and specifies the parameter that is associated with <define-identifier>.

If more than one parameter is specified, the left-to-right order in which the parameters appear in <text> must be the same left-to-right order in which the parameters appear in the parentheses ( ) or bracket [ ] characters. The number of parameters in <text> must equal the number of parameters in the parentheses or bracket characters.

The maximum number of parameters allowed is eight per <define-identifier>.

AS

The keyword AS specifies that the first number sign (#) text-delimiter character is to follow.

#

The number sign (#) characters specify the delimiters of <text>.

text

This field is the text portion of the define statement that contains any SDL/UPL symbol including semicolons, but not the number sign (#) or percent sign (%) characters. The number sign (#) character is the end-of-text delimiter and the percent sign (%) character indicates that the remainder of the source-image record is a comment. Specifying comments within the virgule asterisk and asterisk virgule (/* <comments> */) characters is allowed and the comment is not copied at invocation time.

A maximum of 1024 characters can appear in <text>, excluding comments and superfluous blanks. Also, no unpaired parentheses or brackets can appear in <text>.

All identifiers specified in <text> must be declared prior to an invocation of <define-identifier> and need not be declared prior to the define statement.

Example 1:

```
DEFINE PROC AS #PROCEDURE#;            % The SDL/UPL compiler replaces
                                       % every occurrence of identifier
                                       % PROC with PROCEDURE.
```

Example 2:

```
DEFINE COMPARE (X,Y) AS                % When the SDL/UPL compiler
    # IF X < Y THEN @(1)1@;            % encounters COMPARE (P1, P2);
               ELSE @(1)0@ #;          % in a source statement, the
                                       % following text is substituted.
                                       %   IF P1 < P2 THEN @(1)1@;
                                       %              ELSE @(1)0@;
                                       % The parameters P1 and P2 in the
                                       % the define statement are
                                       % interpreted as procedure
                                       % parameters.
```

Example 3:

```
DEFINE REPEAT AS #ABC (TAGA, X) #;     % The source statement contained
    .                                  % between the number sign (#)
    .                                  % characters is copied into the
    .                                  % SDL/UPL program whenever the
IF X EQL 9 THEN REPEAT;                % identifier REPEAT is specified.
                                       % The IF statement invokes the
                                       % define statement.
```

Example 4:

```
DEFINE TRIAL (A, B, C) AS              % This statement generates the
  # IF (A) EQL ZERO THEN A := B;       % IF statement whenever the
                     ELSE C #;         % identifier TRIAL is specified.
```

Example 5:

```
DEFINE TRUE  AS # @(1)1@ #,            % The identifiers TRUE and FALSE
       FALSE AS # @(1)0@ #;            % become boolean bit strings
                                       % equal to 1 and 0, respectively.
```

Example 6:

```
DEFINE MAX AS # & IF S1 A := X;        % This statement is available
                & ELSE A := Y;         % to the SDL/UPL compiler but
                & END #;               % only A := X or A := Y is
                                       % compiled, depending on the
                                       % conditional symbol S1.  If
                                       % the statement & SET S1 has
                                       % been encountered, A := X; is
                                       % used.  If S1 has not be set,
                                       % or the & RESET S1 has been
                                       % encountered, then A := Y is
                                       % used.
```

Example 7:

```
DEFINE A AS # IF X GTR 10              % The two statements that follow
                THEN PROCX #,          % the define statement expand to
       C(M) AS # X := M;               % the following:
                A #;                   %   X := Z;
      .                                %   IF X GTR 10 THEN PROCX;
      .                                %   BUMP I BY (R + S);
      .
C(Z);
BUMP I BY (R + S);
```

Example 8:

```
DEFINE MAX_SIZE AS                     % IF a conditional compiler
  # & IF DATACOMM 64                   % control option & SET DATACOMM
    & ELSE 32                          % is specified, the define
    & END #;                           % identifier MAX_SIZE is replaced
                                       % by the number 64. If & SET
                                       % DATACOMM is not specified or &
                                       % RESET DATACOMM is specified,
                                       % MAX_SIZE is replaced by the
                                       % number 32.
```

SECTION 6
EXPRESSIONS

Expressions are the operational portions of statements. If a statement is analogous to a sentence, then expressions are the words and phrases within a sentence. All operational functions, such as comparison, arithmetic, and others, take place within expressions. Exceptions being the assignment and the regular procedure-call functions.

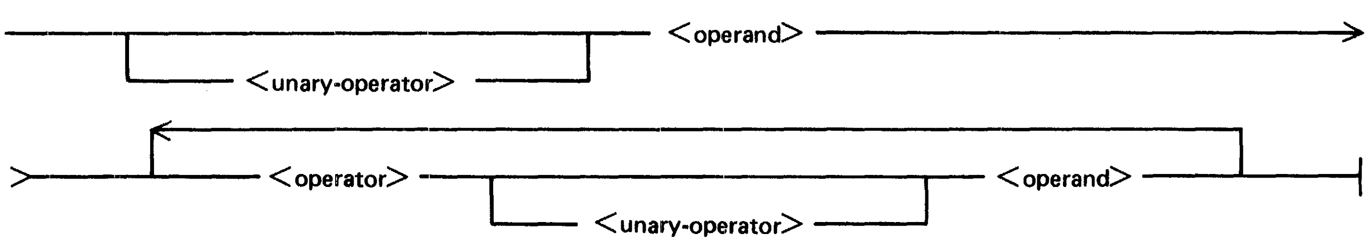
The format of an expression is similar to the format of an algebraic expression. Operators, such as +, −, *, /, and so forth, are used as "infix" notation. Also, parentheses can be used to group the order of evaluation. Each operand can be prefixed with a unary operator.

An expression is defined as recursive and can contain as many operands and operators as are required to produce the desired result.

Expressions are evaluated by performing the indicated operations in a left-to-right order. The sequence in which the operations are performed is determined by the rules of operator precedence. The rules of operator precedence are described in Order of Precedence in this section. When operators have the same precedence, the sequence of operation is determined by the order of the appearance, from left to right. Parentheses can be specified to modify the normal hierarchical sequence of evaluation. An expression within parentheses is evaluated and its value is used in subsequent operations.

The syntax and semantics of an expression are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

unary-operator
: This field can be any valid SDL/UPL unary operator. The unary operators are + (plus) and − (minus).

operand
: This field can be any valid SDL/UPL literal or identifier.

operator

This field can be any valid SDL/UPL operator. The valid SDL/UPL operators follow.

Operator	Function
:=	replace, delete left part
::=	replace, delete right part
+	addition
−	subtraction
*	multiplication
/	division
MOD	remainder
=	equal
EQL	equal
/=	not equal
NEQ	not equal
>	greater than
GTR	greater than
>=	greater than or equal
GEQ	greater than or equal
<	less than
LSS	less than
<=	less than or equal
LEQ	less than or equal
NOT	not
AND	and
OR	or
EXOR	exclusive-or
CAT	concatenation

UNARY OPERATORS

The following are the unary operators.

Operator	Function
+	plus
−	minus

The unary operator acts upon one operand. It can never appear as an infix operator between two operands. It can appear to the right of any other operator, including itself.

Minus

The unary minus (−) generates the two's complement of the operand associated with it (−X = (NOT X) + 1). The operand can have any data type. If the data type is FIXED, the unary minus has the effect of reversing the sign, and the result is stored on the evaluation stack with a FIXED data type. If the operand is either a character or bit string, only the rightmost 24 bits are evaluated. Character or bit strings less than 24 bits are padded with leading zeroes up to 24 bits. The two's complement of the string is generated and returned to the evaluation stack with a FIXED data type.

Example:

```
X := -1;            % Identifier X is assigned the value of -1.

X := -A;            % Identifier X is assigned the two's complement
                    % of identifier A.
```

Plus

The SDL/UPL compiler generates no code for the unary plus (+). The unary plus exists only for program documentation purposes.

Example:

```
X := +1;            % Identifier X is assigned the value of 1.

X := +A;            % Identifier X is assigned the value of A.
```

ARITHMETIC OPERATORS

The following are the arithmetic operators.

Operator	Function
+	Addition
−	Subtraction
*	Multiplication
/	Division yielding integer value of quotient
MOD	Division yielding integer value of remainder

The arithmetic operators perform 24-bit arithmetic on two operands of any of the three data types. If both operands are declared with FIXED data types, sign analysis is performed. If the operands are not declared with FIXED data types, only the rightmost 24 bits of each operand are used in the evaluation. If an operand has a length less than 24 bits and is declared with a BIT or CHARACTER data type, leading zeroes are padded in the leftmost bits prior to the operation.

The result of an arithmetic operation stores a 24-bit result on the evaluation stack. If both operands are declared with FIXED data types, the result is a FIXED data type. If either operand is declared with other than a FIXED data type, the result is a BIT data type.

Addition

The + (addition) operation causes the values of the two operands to be added.

Examples:

```
X := A + B;              % Identifier X is assigned the sum of
                         % identifiers A and B.

X := 1 + A;              % Identifier X is assigned the sum of
                         % 1 plus the value of identifier A.
```

Subtraction

The – (subtraction) operation causes the value of the right operand to be subtracted from the value of the left operand.

Examples:

```
X := A - B;                   % Identifier X is assigned the value
                              % of identifier A less the value of
                              % identifier B.

X := A - 1;                   % Identifier X is assigned the value
                              % of identifier A less 1.
```

Multiplication

The * (multiplication) operation causes the values of the two operands to be multiplied together.

Examples:

```
X := A * B;                   % Identifier X is assigned the value of
                              % identifier A multiplied by the value
                              % of identifier B.

X := A * 25;                  % Identifier X is assigned the value of
                              % identifier A multiplied by 25.
```

Division

The / (division) operation causes the value of the left operand to be divided by the value of the right operand. Any remainder is truncated.

Examples:

```
X := 7 / 3;          % Identifier X is assigned the value 2.

Y := 3 / 7;          % Identifier Y is assigned the value 0.

Z := A / B;          % Identifier Z is assigned the value of
                     % identifier A divided by the value of
                     % identifier B.
```

The multiplication and division operators do not associate.

Examples:

```
(A * B) / C does not equal A * (B / C)

X := (4 * 5) / 7;             % Identifier X is assigned the value 2.

Y := 4 * (5 / 7);             % Identifier Y is assigned the value 0.
```

MOD

The MOD operation is the modular operation. A modular operation is the value that is left (remainder) after a division operation is performed. The following formula is used in performing a MOD operation where a and b are any operands.

a MOD b = a − (b * (a / b))

Examples:

```
A := 7 MOD 3;      % Identifier A is assigned the value equal to
                   % 7 - (3 * (7 / 3)) = 7 - (3 * 2) = 1.

B := -7 MOD 3;     % Identifier B is assigned the value equal to
                   %  -7 - (3 * (-7 / 3)) = -7 - (3 * (-2) = -1.

C := 3 MOD -7;     % Identifier C is assigned the value equal to
                   % 3 - ((-7) * (3 / (-7))) = 3 - ((-7) * 0) = 3.

D := -3 MOD -7     % Identifier D is assigned the value equal to
                   % (-3) - ((-7) * ((-3) / (-7))) = (-3) - ((-7) * 0
                   %  = 3.
```

Negative arguments do not follow the traditional definitions of modular arithmetic in mathematics.

RELATIONAL OPERATORS

The following are the relational operators.

Operator	Function
=	equal
/=	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
EQL	equal to
NEQ	not equal
GTR	greater than
LSS	less than
GEQ	greater than or equal
LEQ	less than or equal

The relational operators cause a comparison operation between two operands of any data type. If the comparison is TRUE, the 1-bit result, @(1)1@, is returned. If the comparison is FALSE, the 1-bit result, @(1)0@, is returned.

If both operands are declared with FIXED data types, the operator does a true-sign comparison. If both operands are character strings, the shorter operand is padded on the right with blanks and a character-by-character comparison using the EBCDIC collating sequence is performed. For all other operand combinations, leading zeroes are padded into the leftmost bits of the shorter operand. No sign analysis is performed and the operands are treated as positive values.

Examples:

```
X := 1 = 2;       % Identifier X is assigned the value @(1)0@.
X := 1 /= 2;      % Identifier X is assigned the value @(1)1@.
X := 1 > 2;       % Identifier X is assigned the value @(1)0@.
X := 1 GEQ 2;     % Identifier X is assigned the value @(1)0@.
X := 1 LSS 2;     % Identifier X is assigned the value @(1)1@.
X := 1 LEQ 2;     % Identifier X is assigned the value @(1)1@.
```

LOGICAL OPERATORS

The following are the logical operators.

Operator	Function
NOT	not
AND	and
OR	or
EXOR	exclusive-or

The logical operators perform a bit-by-bit analysis on all three data types. The NOT logical operator is considered a unary operator and can appear to the right of any other operator (including itself). The result of each logical operator for every boolean value of X and Y is summarized in Table 6-1.

Table 6-1. Boolean Logic Table

Boolean Value		Result				
X	Y	NOT X	NOT Y	X AND Y	X OR Y	X EXOR Y
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	1	0

Example:

```
DECLARE  (A, B, C, D, X, Y) BIT(8);
X := @(1)00101110@;
Y := @(1)10101100@;

A := NOT X;     % Identifier A is assigned the value @(1)11010001@.

B := X AND Y;   % Identifier B is assigned the value @(1)00101100@.

C := X OR Y;    % Identifier C is assigned the value @(1)10101110@.

D := X EXOR Y;  % Identifier D is assigned the value @(1)10000010@.
```

CAT OPERATOR

The CAT operator is a concatenate operator that joins two strings of data and forms a new string. Any combination of data types or data strings can be concatenated. The resultant string cannot exceed 8191 characters or 65,535 bits.

Character string concatenation is the most common concatenation operation. If two strings to be concatenated are character strings, the result is a character string. Concatenation of any other combinations of data types results in a bit string.

Example:

```
DECLARE A       CHARACTER,
        B       BIT (3),
        C       FIXED,
        X       BIT (6),
        Y       CHARACTER (2),
        Z       BIT (11),
        XX      BIT (27);

A := "B";            % Identifier A comprises a character string
                     % containing the letter B.

B := @(1)101@;       % Identifier B comprises a bit string that
                     % contains the binary value of five.  The
                     % length of the identifier is three bits.

C := 10;             % Identifier C comprises a fixed string that
                     % contains the positive (+) decimal value of 10.

X := B CAT B;        % A binary value of 45 or @(1)101101@ is
                     % created.  The length of the data string
                     % is six bits and the result of the
                     % concatenation is assigned to the identifier
                     % X.
```

```
Y := A CAT A;            % A character string, comprised of two bytes,
                         % that has a value of "AA" is created.  This
                         % value is assigned to the identifier Y.

Z := A CAT B;            % A binary value of 1557 or @(1)11000010101@
                         % is created.  The length of the data string
                         % is 11 bits.  The result of the concatenation
                         % is assigned to the identifier Z.

XX := B CAT C;           % A binary string equivalent to the SDL/UPL
                         % octal notation @(3)5000000012@ is created.
                         % The result of the concatenation is assigned
                         % to the identifier XX.

X := A CAT B := 4;       % The CAT operator is lower in precedence than
                         % the := assignment operator.  Identifier B
                         % is set to a value of four before identifier
                         % B is concatenated with identifier A.  The
                         % result of the concatenation is then
                         % assigned to the identifier X.
```

Example Program:

```
DECLARE
  01 TIME_OF_DAY                 BIT (72),
     03 HOURS                    BIT (16),
     03 MINUTES                  BIT (16),
     03 SECONDS                  BIT (16),
     03 TENTHS_OF_SECONDS        BIT (8),
     03 AM_OR_PM                 BIT (16);

TIME_OF_DAY := TIME (CIVILIAN, CHARACTER);
DISPLAY ("THE CURRENT TIME IS " CAT HOURS CAT ":" CAT MINUTES
         CAT ":" CAT SECONDS CAT "." CAT TENTHS_OF_SECONDS CAT
         " " CAT AM_OR_PM);
STOP;
FINI;

% This example program obtains the current time from the MCP,
% displays the hours, minutes, seconds, tenths of a second, and
% AM or PM on the ODT.  The CAT operator verb is used to concatenate
% the message.
```

Output from Example Program:

```
% TESTO =2403 THE CURRENT TIME IS 12:35:16.0 PM
```

CONDITIONAL EXPRESSION

The conditional operator expression uses the keywords IF, THEN and ELSE or the CASE verb. Refer to Section 9 for a complete description of IF, THEN and ELSE keywords and the CASE verb.

REPLACEMENT OPERATORS

The following are the replacement operators.

Operator	Function
:=	delete left
::=	delete right

The replacement operation is performed within an expression and evaluation continues after the replacement is made.

Delete Left (:=)

The delete-left operator assigns the value of the operand on the right to the operand on the left. The new value of the operand on the left remains on the evaluation stack without any change to its attributes. Any truncation or realignment of data that takes place during the replacement is not reflected during evaluation of the expression.

Example:

```
DECLARE CC    CHARACTER (2),
        BB    BIT (4),
        AA    CHARACTER (2);

AA := BB := CC := "6";
```

The following describes the action taken to evaluate the example.

1. The value being assigned is the literal "6" (@F6@).
2. The value@F6@is stored,left-aligned,into identifier CC.It is padded on the right with a blank @40@ character, because identifier CC has a data type equal to CHARACTER and is longer than @F6@. The resulting value of identifier CC is @F640@.
3. The value @F6@ is stored, right-aligned with truncation, into identifier BB, because identifier BB has a data type equal to BIT and is shorter than @F6@. The resulting value of identifier BB is @6@.
4. The value @F6@ is stored left-aligned into identifier AA and is padded on the right with a blank @40@ character, because identifier AA has a data type equal to CHARACTER and is longer than @F6@. The resulting value of identifier AA is @F640@.

Figure 6-1 shows the status of the evaluation stack and each identifier as the evaluation of AA := BB := CC := "6" is performed.

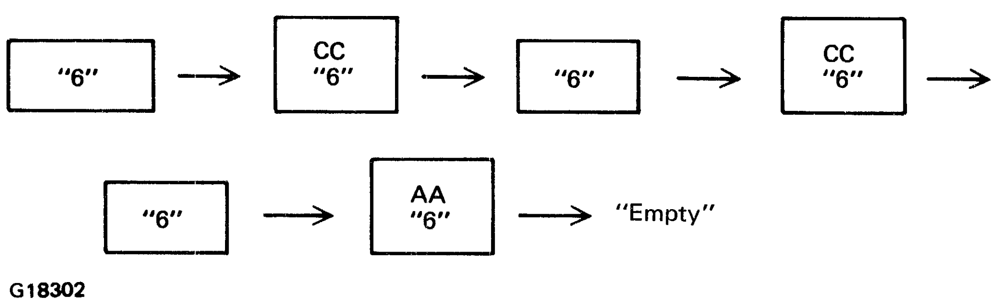


Figure 6-1. Status of the Evaluation Stack

Delete Right (:: =)

The delete-right (:: =) operator evaluates the operand to the right and stores the value into the memory location referenced by the operand to the left. The value of the operand to the right becomes unavailable during any further evaluations. The continued evaluation of the operands uses the value and attributes of the operand to the left of the operator. Any truncation or realignment of data that takes place during the replacement is reflected during the continued evaluation of the expression.

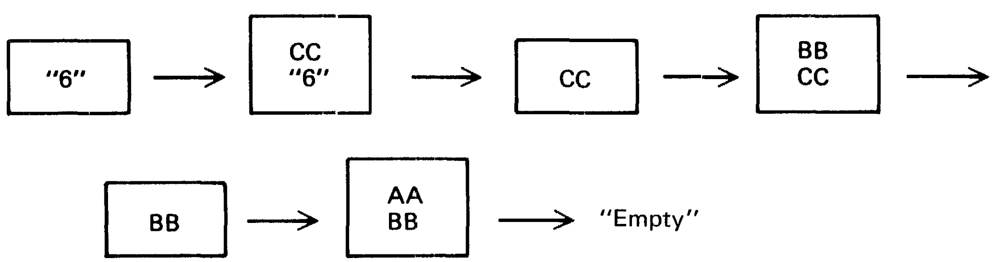
Example:

```
DECLARE   CC   CHARACTER (2),
          BB   BIT (4),
          AA   CHARACTER (2);

AA :=  BB ::=  CC ::=  "6";
```

The following describes the action taken to evaluate the example.

1. The value being assigned is the literal "6" (@F6@).
2. The value@F6@is stored,left-aligned,into identifier CC and is padded on the right with a blank @40@ character, because identifier CC has a data type equal to CHARACTER and is longer than @F6@. The resulting value of identifier CC is @F640@.
3. The value of identifier CC (@F640@) is stored, right-aligned with truncation, into identifier BB since identifier BB has a data type equal to BIT and is shorter than @F640@. The resulting value of identifier BB is @0@.
4. The value of identifier BB (@0@) is stored, right-aligned into identifier AA and is padded on the left with binary zeros @000@, because identifier BB is a bit string. The resulting value of identifier AA is @0000@.

Figure 6-2 shows the status of the evaluation stack and each identifier as the evaluation of AA := BB :: = CC ::= "6" is performed.



G18303

Figure 6-2. Status of the Evaluation Stack

Replacement Operations in Procedures

The following is an example of a delete left and a delete right replacement in a procedure.

Examples:

```
PROCEDURE GOOD BIT VARYING;
  DECLARE X BIT (48);
    RETURN X ::= "RESULT";
END GOOD;

PROCEDURE BAD BIT VARYING;
   DECLARE Y BIT (48);
   RETURN Y := "RESULT";
END BAD;
```

Procedure GOOD returns a bit string, because identifier X remains on the evaluation stack after being evaluated and the data type of identifier X matches the procedure data type of BIT VARYING.

Procedure BAD returns a character string as the result, because identifier Y is deleted from the evaluation stack after being evaluated. The character string "RESULT," which remains on the evaluation stack, does not match the procedure's data type of BIT VARYING. If the FORMAL_CHECK compiler option is specified, procedure BAD produces a run-time error.

ORDER OF PRECEDENCE

The following is the relative binding power (precedence) of the SDL/UPL operators. The operators are listed from highest to lowest order.

+, − (unary operators)
*, /, MOD
+, − (additive operators)
=, /=, >, <, >=, <=
NOT
AND
OR, EXOR
CAT
CASE
IF-THEN-ELSE
Replacement

Refer to Section 9 for a complete description of CASE and IF, THEN, and ELSE.

The replacement operators have higher precedence than any operator to their left and lower precedence than any operator to their right.

The order of evaluation of operators having equal precedence is always from left to right within the expression.

Parentheses and brackets force the enclosed expression to be evaluated completely before any operations outside the parentheses or brackets are evaluated. When parentheses or brackets are nested, the inner-most pair is evaluated first. Within the parentheses or brackets, normal rules of precedence are in effect.

ADDRESS GENERATORS

An address generator includes any expression that leaves an address on the top of the evaluation stack.

The following is the syntax of address generators.

BUMP <identifier> BY <expression>
DECREMENT <identifier> BY <expression>
IF <expression> THEN <identifier> ELSE <identifier>
CASE <expression> OF (<identifier-1>, ... ,<identifier-n>)
<identifier-1> := <identifier-2>
<identifier-1> ::= <expression>

INDEXING (SDL PROGRAMS ONLY)

There are two methods of indexing in an SDL program. They are:

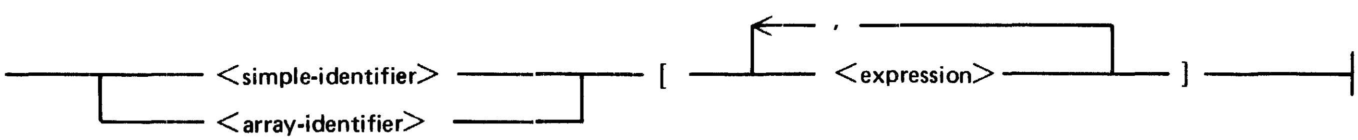
1. The descriptor provides the address and the index provides the offset from this address.
2. The descriptor provides the offset and the index provides the address.

The indexing operation causes the following three events to occur.

1. The simple or array descriptor is loaded to the top of the evaluation stack.
2. If the descriptor is an array descriptor, it is converted to a simple descriptor which describes the first (zero) element of the array.
3. The address field of the descriptor is modified by adding the index to it.

Self-relative data items cannot be indexed. For example, data items whose length is not greater than 24 bits, are not in a structure, and do not remap some other data item.

SDL Syntax:



Syntax Semantics:

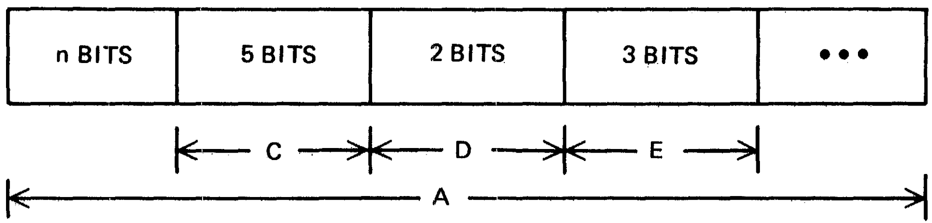
simple-identifier
This field can be any valid SDL identifier with a length greater than 24 bits, and specifies the name of the template used for indexing.

array-identifier
This field can be any valid SDL array identifier and specifies the name of the template used for indexing.

expression
This field can be any valid SDL expression and specifies the offset to be used for indexing. If more than one <expression> is specified, the sum of the expressions is used.

Example:

Assume the following is a memory layout of an SDL program and identifier N has the value of n (the offset from the beginning of identifier A to identifier B). Identifier D can be accessed using either of the two methods.



Method 1:

```
DECLARE   01 A            BIT (5000),
             03 B,
                05  C     BIT (5),
                05  D     BIT (2),
                05  E     BIT (3),
          N               BIT (24),
          X               BIT (2);

X := D [N];      % This statement moves identifier D (with the offset
                 % given by identifier N) into identifier X.
```

Method 2:

```
DECLARE   A               BIT (5000),
          01   B   REMAPS BASE,
               03   C    BIT (5),
               03   D    BIT (2),
               03   E    BIT (3),
          N              BIT (24),
          X              BIT (2);

X := D [N, DATA_ADDRESS (A)];     % This statement moves identifier
                                  % D (with the offset given by the
                                  % sum of identifier N and
                                  % DATA_ADDRESS (A)) into identifier
                                  % X.
```

NOTE

The following must be noted concerning method 2.

- The structure of identifiers B, C, D, and E, which remaps base is called a "template".
- This template can be applied to any data area by providing the address part of the index. This is not the case when method 1 of indexing is used.
- If identifier N contained the address of identifier B rather than the offset to identifier B from the beginning of identifier A, then the statements which assign identifier D into identifier X are identical (X := D [N];).

SECTION 7
PROCEDURES

Procedures are the basic program structure in an SDL/UPL program. Each is a self-contained functional unit within the program.

This section is divided into four parts. These parts are Procedure Declaration Statement, Procedure Body, Procedure End Statement, and Procedure Invocations.

PROCEDURE DECLARATION STATEMENT AND PARAMETERS

The PROCEDURE declaration statement specifies the beginning of a new procedure and is optionally followed by parameters enclosed with the parenthesis "( )" characters.

Specifying a parameter in the procedure declaration statement allows the procedure to reference values of identifiers that are outside the global range of the procedure. A parameter is a local identifier of the procedure.

Every parameter specified in the procedure declaration must have an associated FORMAL or FORMAL_VALUE declaration.

FORMAL declarations must be separate statements from FORMAL_VALUE declarations.

The data types of formal and formal-value parameters should match the data types of the corresponding actual parameters. The SDL/UPL compiler does not automatically check to ensure that these match. If the compiler control option FORMAL_CHECK is set, data types are checked at run-time.

Varying formal parameters can be remapped. If a varying formal parameter is remapped, the parameter and its corresponding actual identifier must meet the remap restrictions. A warning message is generated by the SDL/UPL compiler when a formal parameter is remapped.

Formal parameter arrays can be given a variable number of elements by specifying the asterisk (*) character within the parentheses characters in the formal declaration.

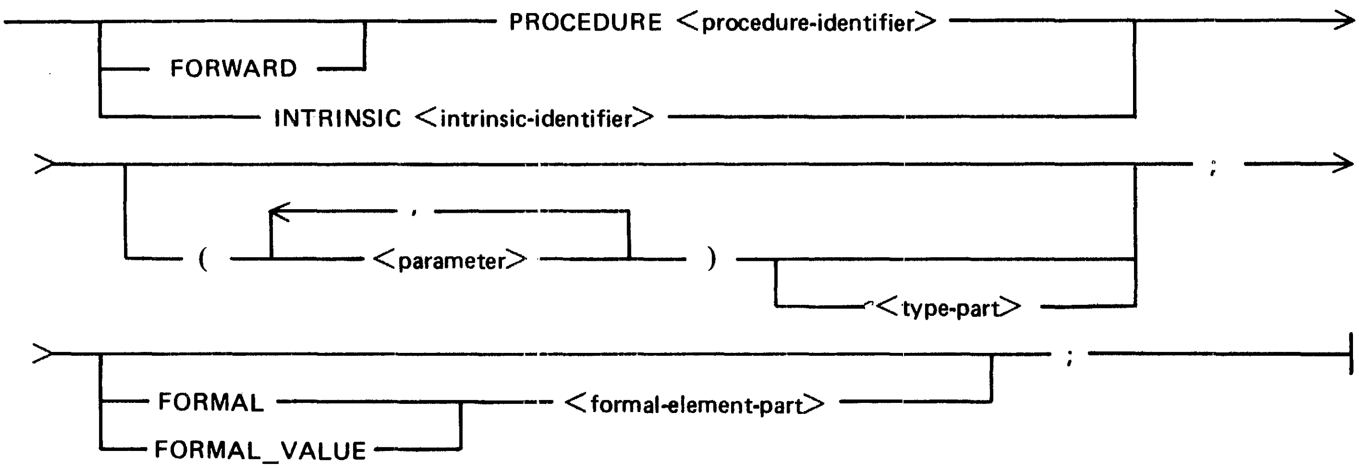
Example:

```
PROCEDURE X (A);
FORMAL A(*) FIXED;
```

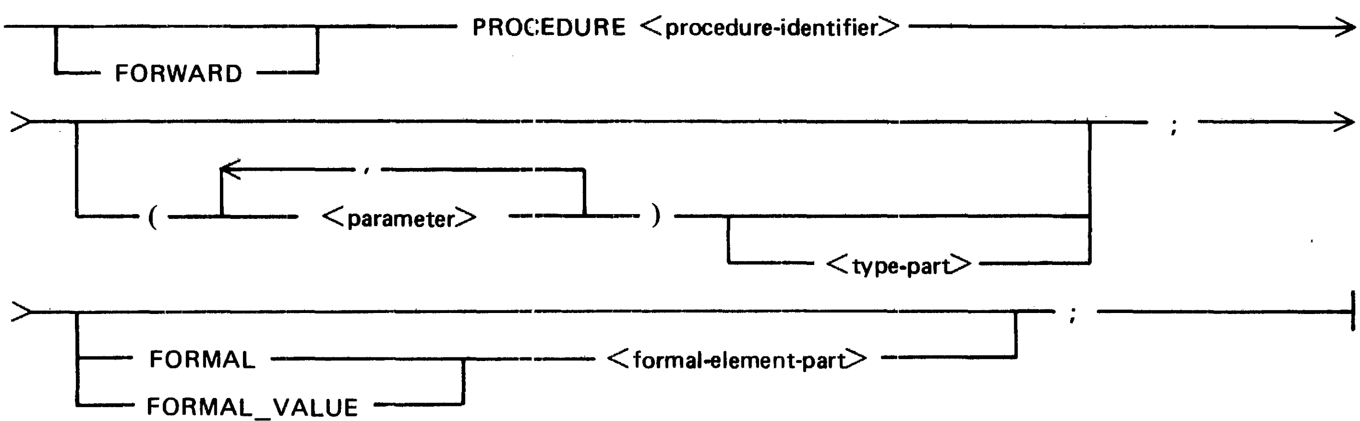
A level-structured identifier can be passed by naming only the 01 level of the structure. The subfields of the structure do not remain defined when the structure is passed to a procedure. Any attempt to remap the parameter generates a syntax error.

The syntax and semantics of the PROCEDURE declaration are described as follows:

SDL Syntax:



UPL Syntax:



Syntax Semantics:

FORWARD

The keyword FORWORD causes the procedure to be a forward procedure.

Before a procedure can be invoked, it must be declared. A problem can arise when one procedure invokes another procedure which in turn invokes the first. In this case, whichever procedure appears first must contain at least one reference to the second procedure which has not yet been declared. The FORWARD keyword allows the use of forward and recursive references by providing a temporary procedure declaration.

The FORWARD PROCEDURE statement does not eliminate the need for the normal procedure declaration which must follow in the program.

The FORWARD PROCEDURE statement must be in the same scope as its associated procedure and it must be specified immediately prior to or after the declarations.

The return data type must also be declared in the FORWARD PROCEDURE statement.

When the FORWARD PROCEDURE statement refers to a procedure with parameters, it must include those parameters in the FORWARD PROCEDURE declaration. Also, any FORMAL declaration statement of the parameters must accompany the FORWARD PROCEDURE statement. Also, the formal declarations must appear within the actual procedure.

INTRINSIC

The keyword INTRINSIC is used only by SDL programs and causes the file specified by <intrinsic-identifier> to be included. The intrinsic must begin at displacement 0 in a new segment.

intrinsic-identifier

This field can be any valid SDL intrinsic file name and specifies the intrinsic file to use.

PROCEDURE

The keyword PROCEDURE is required for a procedure declaration.

procedure-identifier

This field can be any valid SDL/UPL identifier and specifies the name of the procedure.

parameter

This field can be any valid SDL/UPL identifier and specifies the identifier that is used and not declared in the procedure. If <type-part> follows <parameter>, the value of <parameter> is returned to the statement that invoked the procedure. If there is no <type-part> specified, the value of <parameter> is passed from the statement that invokes the procedure. If this field is specified, a FORMAL or FORMAL_VALUE statement must immediately follow the procedure statement.

type-part

Refer to type-part later in this section.

Procedures which return explicitly a value when completed are called "typed" procedures. The data type of the returned value must be specified in the procedure declaration.

If the data type of the returned value does not match the specified data type, an advisory message is generated by the SDL/UPL compiler during compilation.

FORMAL

When a parameter is specified in the procedure declaration and when it is desirable to have the corresponding identifier's value changed, the keyword FORMAL is required, provided that any change to the value of <parameter> is made in the procedure.

When a parameter is declared with the FORMAL keyword, the parameter refers to the address of the actual identifier. This requires that the parameter correspond to an identifier. All changes made to <parameter> are made to the actual identifier.

If the parameter in the FORMAL part of the procedure declaration is an array, then only an array can be passed to the procedure. If an array is to be passed to a procedure as a parameter, the corresponding FORMAL declaration of the procedure must specify an array.

FORMAL_VALUE

When a parameter is specified in the procedure declaration and when it is not desirable to have the value of the corresponding identifier changed, the keyword FORMAL_VALUE is required, provided that any change to the value of <parameter> is made in the procedure.

When <parameter> is declared with the FORMAL_VALUE keyword, <parameter> receives the value of the actual identifier. This identifier must yield a value. It can be a literal, a number, or an identifier enclosed in the quotation mark(")characters. The quoted identifier "<identifier>" notation forces references to the value rather than the address of the identifier. Changes to the formal-value parameter are known only within the scope of the procedure in which the formal-value parameter is declared.

When the name (address) of an identifier is passed to a formal-value parameter, the value of the actual identifier is assigned to the formal-value parameter. Changes made to the formal-value parameter are not reflected in the corresponding actual identifier.

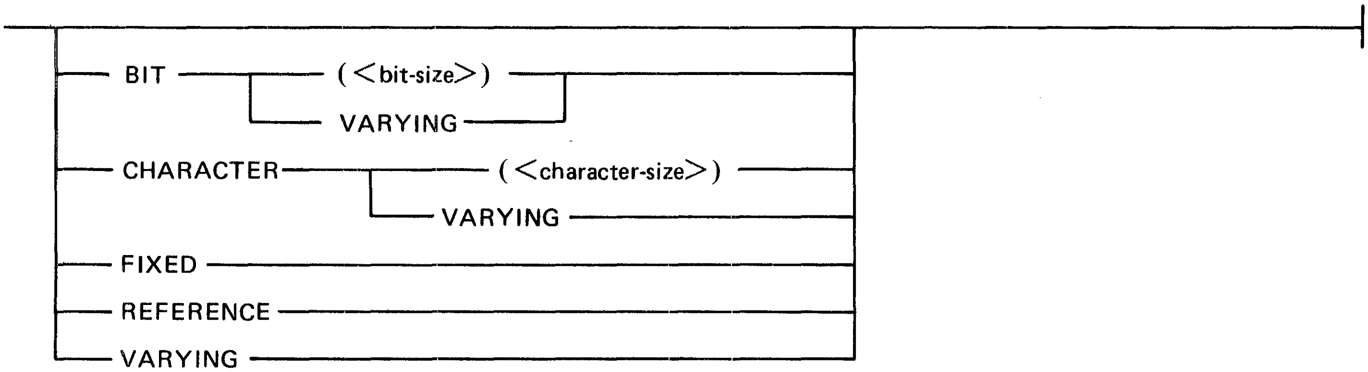
formal-element-part

Refer to formal-element-part in this section.

type-part

The syntax and semantics of the type-part of the PROCEDURE declaration are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

BIT
: The keyword BIT in the procedure declaration specifies that the value of the parameter to be returned from the procedure has a data type equal to BIT.

  The keyword BIT in the formal declaration specifies that the data type of <parameter> passed to or returned from the procedure has a data type equal to BIT.

CHARACTER
: The keyword CHARACTER in the procedure declaration specifies that the value of the parameter to be returned from the procedure has a data type equal to CHARACTER.

  The keyword CHARACTER in the formal declaration specifies that the data type of <parameter> passed to or returned from the procedure has a data type equal to CHARACTER.

FIXED
: The keyword FIXED in the procedure declaration specifies that the value of the parameter to be returned from the procedure has a data type equal to FIXED.

  The keyword FIXED in the formal declaration specifies that the data type of <parameter> passed to or returned from the procedure has a data type equal to FIXED.

REFERENCE
: The keyword REFERENCE in the procedure declaration specifies that the value of the parameter to be returned from the procedure has a reference identifier.

  The keyword REFERENCE in the formal declaration specifies that the data type of <parameter> passed to or returned from the procedure has a data type of a reference identifier.

VARYING

The keyword VARYING in the procedure declaration specifies that the value of the parameter to be returned from the procedure can vary in data type and length.

The keyword VARYING in the formal declaration specifies that the data type of <parameter> passed to the procedure can vary in data type and length.

If the keyword VARYING follows the keywords BIT or CHARACTER, the length of the bit or character parameter can vary.

bit-size

This field can be any valid SDL/UPL number or expression that generates a value at compilation time and specifies the length in bits of the parameter.

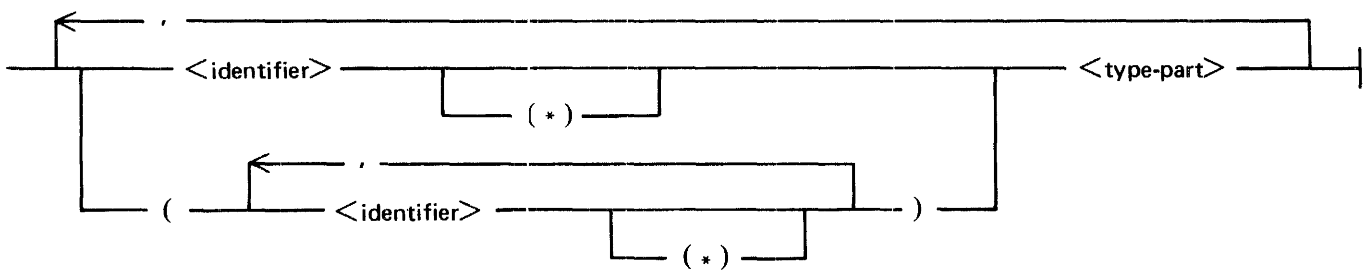
character-size

This field can be any valid SDL/UPL number or expression that generates a value at compilation time and specifies the length in characters of the parameter.

formal-element-part

The syntax and semantics of the formal-element-part of the PROCEDURE declaration are described as follows:

SDL and UPL Syntax:



Syntax Semantics:

identifier

This field can be any valid SDL/UPL identifier and specifies the name of the field whose address or value is passed to the procedure.

(*)

The asterisk character between the parenthesis characters specifies that the number of elements in the array specified by <identifier> can vary when the array is passed to the procedure.

If the parameter in the FORMAL part of the procedure declaration is an array, only an array can be passed to the procedure. If an array is to be passed to a procedure as a parameter, the corresponding FORMAL declaration of the procedure must specify an array.

type-part

Refer to type-part in this section.

The data type of the identifier which is passed to the procedure is specified by type-part.

Example 1:

```
PROCEDURE XYZ;                     % Procedure identifier XYZ
   .                               % is declared.
   .
   .
END XYZ;
```

Example 2:

```
FORWARD PROCEDURE X;               % Procedure identifier X is
   .                               % being declared as a forward
   .                               % procedure.  It can be invoked
   .                               % after this procedure
END X;                             % declaration and before the
                                   % procedure is encountered by
                                   % the SDL/UPL compiler.
```

Example 3:

```
PROCEDURE ABC (X, Y, Z);           % Procedure identifier ABC has
FORMAL X       FIXED,              % three parameters that must be
       Y       CHARACTER VARYING,  % declared formally.  Parameter
       Z (*) BIT VARYING;          % X is an identifier with a data
     .                             % type equal to FIXED.  Parameter
     .                             % Y is an identifier with a data
     .                             % type equal to CHARACTER and
END ABC;                           % the length is calculated on
                                   % each invocation of procedure
                                   % ABC.   Parameter Z is an array
                                   % identifier with a varying
                                   % number of elements (which are
                                   % calculated on each invocation
                                   % of procedure ABC) and a data
                                   % type equal to BIT.
```

Example 4:

```
PROCEDURE SQUARE (N);              % Procedure identifier SQUARE is
FORMAL N FIXED;                    % invoked from a point in the
   .                               % program. A value for identifier
   .                               % N is passed to the procedure
   .                               % by the invoking statement.
   RETURN;
   .
   .
   .
END SQUARE;
```

Example 5:

```
PROCEDURE CUBE (A, B, C,);          % Two procedures, one nested
FORMAL (A, B, C) FIXED;             % within the other, are declared.
  PROCEDURE SQUARE (N);             % The procedure SQUARE can be
  FORMAL N FIXED;                   % invoked only from within the
       .                            % procedure CUBE.
       .
       .
     IF A THEN RETURN;
       .
       .
       .
  END SQUARE;
   .
   .
   .
  IF B THEN RETURN;
       ELSE DO;
               SQUARE (C);
               RETURN;
            END;
END CUBE;
```

Example 6:

```
PROCEDURE ABSVAL (X) FIXED;         % The function procedure ABSVAL
FORMAL X FIXED;                     % returns the absolute value of
  RETURN (IF X LSS 0 THEN - X       % the parameter passed.  The IF
                     ELSE + X);     % expression within the RETURN
END ABSVAL;                         % statement returns the positive
                                    % value of the parameter.
```

Example 7:

```
PROCEDURE MSG CHARACTER (20);       % The function procedure MSG
  DECLARE DATA CHARACTER (20);      % accepts a message from the ODT
  RETURN (ACCEPT DATA);             % and returns it to the invoking
END MSG;                            % IF statement.
   .
   .
   .
IF SUBSTR (MSG, 0, 3) = "YES"
   THEN ......;
   ELSE ......;
   .
   .
   .
```

PROCEDURE BODY

The procedure body follows the procedure and the formal declaration statement. Declarations of local data, nested procedures, and statements are included in the procedure body.

The RETURN verb takes one of two forms depending on the type of the procedure encompassing it. When a data type is specified for the parameters in the procedure declaration, the procedure is a "typed" procedure. If the procedure is a "typed" procedure, an expression must be returned to the point of invocation. If the procedure is not "typed", the RETURN does not allow an expression. Procedure type-checking on the RETURN verb is performed at run time when the FORMAL_CHECK compiler control option is set.

Within any given procedure, certain statements can be nested within other statements and can be accessed like a procedure by an address generated by the larger statement. The most general nesting level is zero. The nesting level of any statement appears on the SDL/UPL compiler listing under the column NL. The following are the most common instances of statements occurring at nesting level 01 or greater.

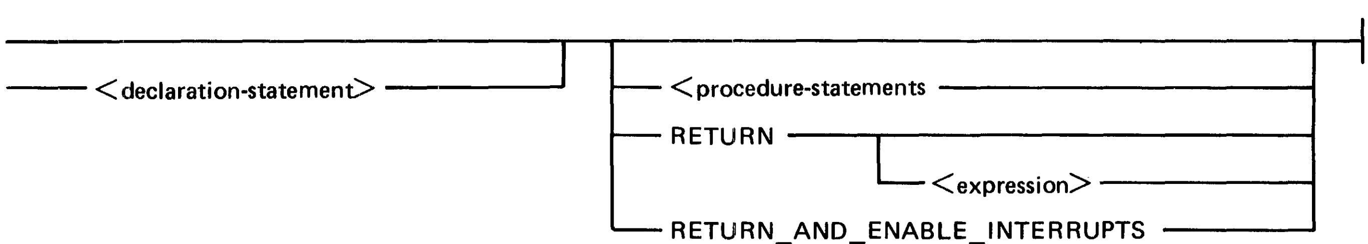
1. The conditional statements following the THEN and ELSE keywords in the IF verb.
2. Statements contained within a CASE group.
3. Statements contained within a DO group.

The SDL/UPL compiler always generates a RETURN statement (even if not specified) directly preceding the END <procedure-identifier>; statement. This ensures that the exit from a procedure is always correct.

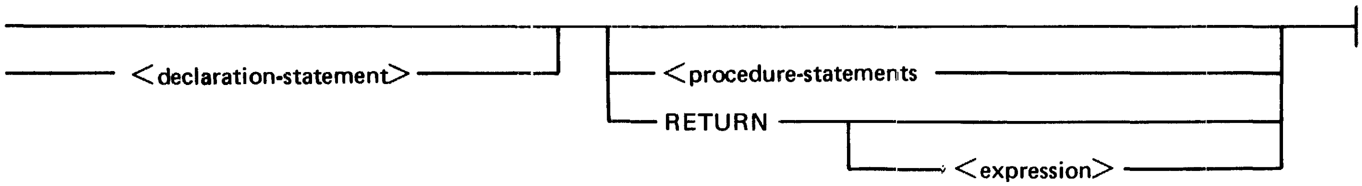
If the procedure is a "typed" procedure, the following value is returned based on the data type of the returned data item.

Data Type to be Returned	**Value Returned**
BIT	Zeros for the length specified
CHARACTER	Blank characters for the length specified
FIXED	Fixed Zero
BIT VARYING	Eight bits of zeros
CHARACTER VARYING	One blank character
VARYING	Fixed zero

SDL Syntax:



UPL Syntax:



Syntax Semantics:

declaration-statement
Refer to Data Declarations in Section 5 for a complete description of <declaration-statement>.

procedure-statements
These statements can be any valid SDL/UPL statements.

RETURN
The keyword RETURN causes the procedure to be exited and to resume program execution at the point where the procedure was invoked.

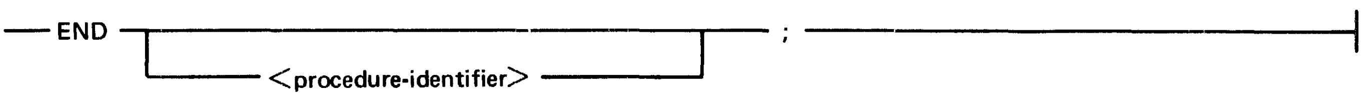
expression
This field can be any valid SDL/UPL expression and specifies the value that is returned to the point where the procedure was invoked.

RETURN_AND_ENABLE_INTERRUPTS
The keyword RETURN_AND_ENABLE_INTERRUPTS is used only by the MCP. This keyword causes a normal procedure exit to occur and enables the interrupt bits.

PROCEDURE END STATEMENT

The procedure end statement follows the procedure body and is the last statement in a procedure.

SDL and UPL Syntax:



Examples:

```
END PROCEDURE_A;

END MAIN_PROCEDURE;
```

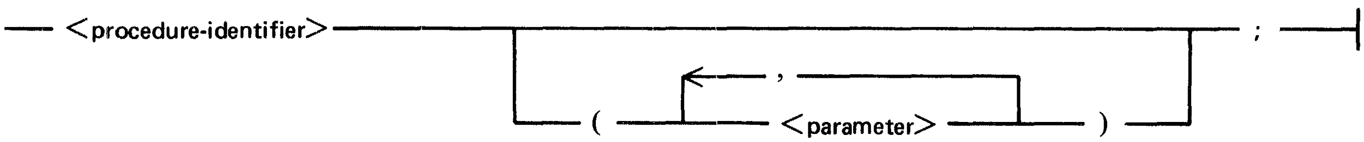
PROCEDURE INVOCATIONS

A procedure is invoked when a procedure identifier is specified in lexic level 0 of the program or in the body of another procedure.

A "typed" procedure invocation produces a value because "typed" procedures return a value. Invoking a "typed" procedure requires that the expected parameters be specified in the procedure invocation. These parameters must be known to the procedure.

Recursive procedure invocations are allowed; that is, a procedure can invoke itself.

SDL and UPL Syntax:



Syntax Semantics:

procedure-identifier

This field can be any valid SDL/UPL procedure identifier that has been declared in a procedure declaration statement. It specifies the name of the procedure to invoke.

parameter

This field can be any valid SDL/UPL identifier that is declared as a parameter in the procedure declaration statement. It specifies the identifier to be passed to or returned from the procedure.

If the parameter in the FORMAL part of the procedure declaration is an array, only an array can be passed to the procedure. If an array is to be passed as a parameter to a procedure, the corresponding FORMAL declaration of the procedure must specify an array.

Example 1:

```
Procedure Declaration      PROCEDURE A;
                              .
Procedure Body                .
                              .
Procedure End              END A;

Procedure Invocation       A;
```

Example 2:

```
Procedure Declaration      PROCEDURE B (J,K,L);
Formal Declaration         FORMAL (J,K) FIXED;
                           FORMAL_VALUE L VARYING;
                                  .
                                  .
                                  .
Procedure End              END B;

Procedure Invocation       B (X, Y, (Z));
```

Example 3:

```
Procedure Declaration     PROCEDURE C (M,N) VARYING;
Formal Declaration        FORMAL M FIXED;
                          FORMAL_VALUE N CHARACTER VARYING;
Procedure Body               DECLARE P FIXED;
                                .
                                .
                                .
                             RETURN (P);
Procedure End             END C;

Procedure Invocation     ANSWER := C(R,S);
```

SECTION 8
STATEMENTS

Statements are the SDL/UPL equivalent of grammatical sentences. They contain a complete sequence of operations (one complete idea). They are logically separate from other similar sequences. While an expression evaluation results in a numerical value, statement evaluation specifies functions or assignments for the values. For example, the expression A + B results in a numerical value and statement X := A + B; (X is replaced by A + B). It assigns the value of the expression to identifier X.

Statements are always terminated by a semicolon (;) character.

Statements fall into three general classifications. These are declaration, control, and assignment statements.

DECLARATION STATEMENTS

Declaration statements connect memory space to identifiers and their attributes. Refer to Section 5 for a complete description of declaration statements.

CONTROL STATEMENTS

Control statements determine the sequence in which statements are executed. They pass control to procedures, bind groups of statements together, or conditionally specify which one of several statements is to be executed next.

Procedure Call Statement

The major control statement in SDL/UPL is the procedure-calling or invoking statement. It consists of a procedure identifier followed by any parameters enclosed in parentheses and terminated by a semicolon (;) character. For example, the procedure ABS, which requires one parameter, is invoked by ABS (VALUE);.

There are three considerations governing the use of procedure-calling statements:

1. A called procedure must be within the scope of the calling statement. In lexic level terminology, a called procedure must be at one of the three following lexic levels.

   a. The procedure can be one lexic level higher and nested within the calling procedure.
   b. The procedure cannot be more than one lexic level lower with a currently invoked procedure that is on an equal or higher lexic level.
   c. The procedure can be a currently invoked procedure on an equal or higher lexic level.

2. A called procedure always returns control back to the calling procedure. There is no GO TO statement in SDL/UPL. The program logic must be structured to use this return-control action. The immediately succeeding statement in the calling procedure is performed when control is returned.
3. The called procedure must be of the proper class. There are two classes of procedures in SDL/UPL. These are function procedures and non-function procedures. Function procedures pass back a value to the function-procedure call and non-function procedures do not.

DO Statements

The DO statement provides the capability to group a set of related statements together for programmatic control purposes. A DO statement consists of the DO statement, optionally followed by <group-name> and/or the FOREVER keyword, and terminated with the semicolon (;) character. The END statement consists of the END statement, optionally followed by <group-name>, and terminated with the semicolon (;) character. The UNDO statement consists of the UNDO statement, optionally followed by <group-name>, and terminated with the semicolon (;) character.

A DO-group consists of a DO statement, one or more executable statements, and an END statement. A DO-group is regarded as a single statement.

A set of DO-groups can be nested. Overlapping DO-groups are not allowed. Every END statement is paired with the preceding unmatched DO statement, starting at the innermost set. An END statement is required for each DO statement. DO-groups can be imbedded in CASE statements, IF statements, or other DO-groups. A maximum of 32 CASE statements, IF statements, or DO-groups can be imbedded in one DO-group. However, the UNDO statement only exits up to a maximum of 16 nested DO-groups. A maximum of 11 levels of labeled DO statements are allowed in an SDL/UPL program.

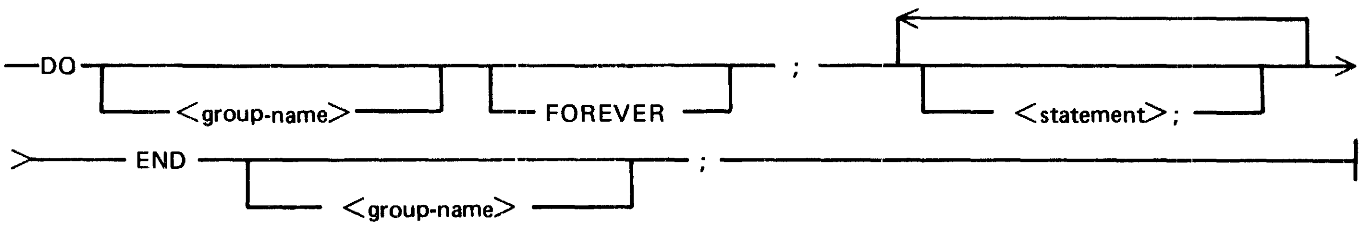
DO-groups, IF statements, and CASE statements define a source-code nesting level that is placed under the column marked NL on the compiler-generated source listing. Each nest must be wholly contained within its outer nest. That is, source-code nesting levels cannot overlap.

The keyword FOREVER causes an unlimited number of DO-group iterations. When an UNDO, RETURN, or STOP statement is performed the DO-group is terminated. If an UNDO statement is performed, the innermost or DO-group labeled in the UNDO statement is terminated. If a RETURN statement is performed, an implicit UNDO statement is performed for all nested DO-groups within the procedure and control is passed to the statement that immediately follows the statement that called the procedure. If a STOP statement is performed, the program goes to end of job.

If the keyword FOREVER is not specified, the DO-group is performed only one time.

There is a limit on the size of a DO FOREVER-group. This limit is 4096 bits of object code generated by the SDL/UPL compiler.

SDL and UPL Syntax:



Syntax Semantics:

group-name

This name labels a DO-group and when specified, must immediately follow the DO statement and END statement. For example, DO <group-name>; and END <group-name>;. <group-name> must be the same in the DO-statement (DO <group-name>;) and in the matching END-statement (END <group-name>;).

FOREVER

The keyword FOREVER causes the DO-group to be performed until an UNDO or RETURN statement is performed for this DO-group.

statement

This field can be any valid SDL/UPL statement. There is no actual limit to the number of statements that can be specified in a DO-group. All SDL/UPL statements must end with the semicolon (;) character.

Example 1:

```
DO;                              % The format of a DO-group requires
   BUMP SUM;                     % the DO and a corresponding END
   DECREMENT DIFF;               % statement.
   .
   .
   .
END;
```

Example 2:

```
IF X EQL 0                       % One of the DO-groups within the
     THEN DO;                    % IF statement is executed, and then
             BUMP X;             % control is passed beyond the IF
              .                  % statement.  The second DO-group is
              .                  % named OTHER, and its END statement
              .                  % must also contain the same name.
          END;
     ELSE DO OTHER;
             DECREMENT X;
              .
              .
              .
             BUMP SUM;
          END OTHER;
```

Example 3:

```
DO THIS_ONE FOREVER;             % The DO-group name THIS_ONE
   IF SUM LEQ ZERO               % iterates until SUM is greater than
      THEN DO;                   % 0.  When SUM is greater than 0,
              BUMP SUM;          % the UNDO statement in the ELSE
              DECREMENT X;       % statement terminates the DO-group.
           END;
      ELSE UNDO;
END THIS_ONE;
```

Example 4:

```
PROCEDURE ABC;                    % This procedure contains several
  DO ANY FOREVER;                 % DO-groups.  The RETURN statement
     IF X GEQ 0                   % in the last IF statement terminates
        THEN DO;                  % the DO-group labeled ANY by passing
               DECREMENT X;       % control out of procedure ABC.
               BUMP SUM;
             END;
     IF SUM GEQ 0
        THEN UNDO;
        ELSE RETURN;
  END ANY;
          .
          .
          .
END ABC;
```

Example 5:

```
DO SETA;                          % This is a DO statement that binds
   X := X + 1;                    % three statements to the DO group
   A_PARM := ZERO;                % SETA.
   ROUTINE (X, A_PARM);
END SETA;
```

Example Program:

```
DECLARE
    TIME_ONE          FIXED,
    TIME_TWO          FIXED,
    CORRECT_ANSWER    FIXED,
    ANSWER            CHARACTER (8);

DO MAIN_LOOP FOREVER;

    TIME_ONE := CONVERT (TIME (COUNTER, BIT), FIXED);
    TIME_TWO := CONVERT (TIME (COUNTER, BIT), FIXED);
    DISPLAY ("HOW MUCH IS " CAT
              CONVERT ((TIME_ONE MOD 57829), CHARACTER) CAT " PLUS "
              CAT CONVERT ((TIME_TWO MOD 100000), CHARACTER));
    ACCEPT ANSWER;
    IF ANSWER = "BYE"
    THEN DO;
            DISPLAY ("GOOD BYE");
            STOP;
         END;
    CORRECT_ANSWER := (TIME_ONE MOD 57829) + (TIME_TWO MOD 100000);
    IF CORRECT_ANSWER = CONVERT (ANSWER, FIXED)
    THEN DO CORRECT;
            DISPLAY ("THAT IS CORRECT, WOULD YOU LIKE TO TRY AGAIN?");
            DISPLAY ("ENTER YES FOR AGAIN OR ENTER BYE TO GO TO EOJ");
            DO FOREVER;
                ACCEPT ANSWER;
                IF ANSWER = "BYE"
                THEN DO;
                       DISPLAY ("GOOD BYE");
                       STOP;
                     END;
                IF ANSWER = "YES" THEN  UNDO;
                ELSE DISPLAY ("INCORRECT RESPONSE TRY YES OR BYE");
             END;
         END CORRECT;
    ELSE DO INCORRECT;
            DISPLAY ("YOUR ANSWER IS INCORRECT");
            DISPLAY ("THE ANSWER IS " CAT
                      CONVERT (CORRECT_ANSWER, CHARACTER));
            DISPLAY ("WOULD YOU LIKE TO TRY AGAIN?");
            DISPLAY ("ENTER YES FOR AGAIN OR ENTER BYE TO GO TO EOJ");
            DO FOREVER;
                ACCEPT ANSWER;
                IF ANSWER = "BYE"
                THEN DO;
                       DISPLAY ("GOOD BYE");
                       STOP;
                     END;
                IF ANSWER = "YES" THEN  UNDO;
                ELSE DISPLAY ("INCORRECT RESPONSE TRY YES OR BYE");
             END;
```

```
        END INCORRECT;

END MAIN_LOOP;

FINI;

% This example program illustrates the use of the DO statement.  The
% program asks the operator to enter the sum of two numbers
% displayed on the ODT.  If the sum is correct, the program asks
% if the operator wishes to continue and try another set of
% two numbers.  If the sum is incorrect, the program displays
% the correct number and asks if the operator wishes to continue
% or try another set of two numbers.  If the response to continue
% is YES to both the correct and incorrect numbers, the program
% displays another set of numbers.  If the response is BYE, the
% program goes to end of job.
```

DO FOREVER Statement

The DO FOREVER statement indefinitely performs the statements within the DO-group until an UNDO statement is performed. Or until control is returned from the procedure in which the DO FOREVER statement is imbedded.

Example:

```
DO PRTN FOREVER;
   X := X + 1;
   ROUTINE (X, A_PARM);          % Procedure Call.
   IF X EQL 5 THEN UNDO;         % Test Limit.
   IF X EQL 10 THEN RETURN;      % Return from the current procedure.
END PRTN;
```

IF, THEN, and ELSE Statement

The IF, THEN, and ELSE keywords are used to conditionally perform one or two statements in an SDL/UPL program.

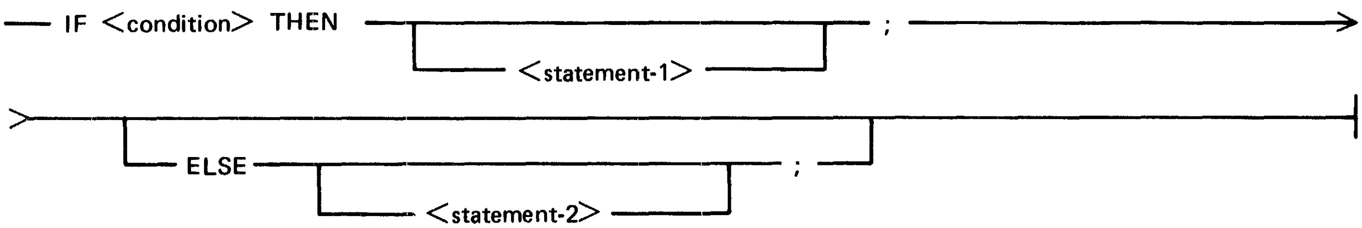
If the rightmost bit of <condition> equals 1, the THEN clause is performed. If the rightmost bit of <condition> equals 0 (zero) and if the ELSE clause is present, the ELSE clause is then performed. Null THEN (THEN;) and ELSE (ELSE;) clauses are allowed. Once the THEN or ELSE (if specified) clause is performed, control is transferred to the next statement. The next statement is the one that immediately follows the THEN clause if no ELSE clause is specified. Or it is the one that immediately follows the ELSE clause, if specified.

If a group of statements are to be performed which are a result of evaluating <condition>, they must be specified in a DO-group that immediately follows the THEN or ELSE keywords. Refer to the DO statement for a complete description on the use of DO-groups.

Nested IF statements are allowed. The maximum number of nested IF statements is 32. The outermost IF-THEN and ELSE are on nesting level 0. <statement-1> and <statement-2> of the IF-THEN and ELSE are on nesting level 1.

The SDL/UPL compiler matches the IF-THEN and ELSE clauses beginning with the innermost nested level. For example, if nesting level 2 has an associated ELSE clause, nesting level 4 must also have an associated ELSE clause.

SDL and UPL Syntax:



Syntax Semantics:

condition
: This field can be any valid SDL/UPL literal, identifier, or expression that returns a value. Only the rightmost bit of <condition> is checked. If the rightmost bit is equal to 1, <condition> is TRUE. If the rightmost bit is equal to 0, <condition> is FALSE.

statement-1
: This statement can be any valid SDL/UPL statement.

statement-2
: This statement can be any valid SDL/UPL statement.

ELSE
: The keyword ELSE causes the statement which immediately follows to be performed if the rightmost bit of <condition> equals 0. Null ELSE clauses (ELSE;) are allowed.

THEN
: The keyword THEN causes the statement which immediately follows to be performed if the rightmost bit of <condition> equals 1. Null THEN clauses (THEN;) are allowed.

Example 1:

```
IF X = 32 THEN Y := 4;      % Identifier Y is assigned a value of 4
                            % if the value of identifier X equals 32.
```

Example 2:

```
IF X > 1 THEN Y := 4;       % Identifier Y is assigned a value of 4
         ELSE Y := 5;       % if the value of identifier X is greater
                            % than 1 and Y is assigned a value of 5
                            % if X is not greater than 1.
```

Example 3:

```
IF X = 1 THEN DO;              % Identifiers Y and Z are assigned the
                    Y := 1;    % values of 1 and 2, respectively, if
                    Z := 2;    % the value of identifier X equals 1.
                END;           % Otherwise, identifiers Y and Z are
            ELSE DO;           % assigned the values 3 and 4,
                    Y := 3;    % respectively.
                    Z := 4;
                END;
```

Example 4:

```
IF X = 2                       % Identifier A is assigned a value of 1 if
THEN IF Y = 3                  % identifier X equals 2, Y equals 3, and
     THEN IF Z = 4             % Z equals 4.  Identifier A is assigned
          THEN A := 1;         % the value of 2 if identifier X equals 2,
          ELSE A := 2;         % Y equals 3, and Z does not equal 4.
     ELSE;                     % Identifiers A and B are assigned the
ELSE IF Y = 20                 % values 3 and 4, respectively, if
     THEN;                     % identifier X does not equal 2 and
     ELSE DO;                  % identifier Y does not equal 20.
              A := 3;
              B := 4;
          END;
```

Example 5:

```
IF A + B GTR X
   THEN DO;
           A := A - 1;
           IF A EQL 0 THEN UNDO;
           FTN_XYZ;
        END;
   ELSE DO;
           X := A + B;
           A := 0;
           B := 0;
        END;
```

Example Program:

```
DECLARE    YES_OR_NO  CHARACTER (3);

DISPLAY ("THIS PROGRAM ILLUSTRATES THE IF, THEN, AND ELSE VERBS.");
DISPLAY ("IF YOU WISH TO CONTINUE, THEN ENTER YES, ELSE ENTER NO");

DO FOREVER;
   ACCEPT YES_OR_NO;

   IF YES_OR_NO = "NO"
   THEN DO;
           DISPLAY ("GOOD BYE");
           STOP;
        END;
   ELSE IF YES_OR_NO = "YES"
        THEN DISPLAY ("YOU ENTERED YES.  IF YOU WISH TO CONTINUE,"
                      CAT " THEN ENTER YES, ELSE ENTER NO.");
        ELSE  DISPLAY ("YES OR NO WAS NOT ENTERED, TRY YES OR NO.");
END;

FINI;
```

CASE Statement

The CASE statement is an expanded form of the IF statement. The evaluation of a conditional expression determines which statement to perform among all the statements associated with the CASE statement. After the statement is performed, control passes to the first statement following CASE statement (if format 2 is specified) or the END CASE statement (if format 1 is specified). If the conditional expression is out of range during program execution, a run time error is generated.

CASE (format-1)

The CASE statement (format-1) selectively performs only one statement within the CASE group of program statements.

At execution time, <index> is evaluated as a binary number. This value is used as a selector to choose from among the program statements in the CASE-group. For example, a value of 2 selects the third program statement. The program statements in the group are numbered from 0 to n-1 for n program statements. A negative value or a value greater than the number of program statements in the CASE-group causes an execution-time error.

All valid SDL/UPL program statements, including nested CASE, DO-group, and IF ... THEN ... ELSE statements, are allowed and are counted as a single statement within the CASE-group of statements.

After the selected program statement is performed, the program performs the program statement immediately following the END CASE; statement.

Null statements can be used to satisfy a program statement position where no operation is to be performed. A null statement is represented by the semicolon (;) character.

If a CASE statement is imbedded in a DO-group and a RETURN verb is specified, the program passes control back to the statement that invoked the procedure.

Each statement within the CASE-group must be an executable statement. If several statements are needed to describe the action to be taken in a given situation, the statements must be blocked in a DO-group. Null statements are allowed.

SDL and UPL Syntax:

```
── CASE <index> ; ──────────────────────────────────────────>
    >───── <statement-0> ; ─────────────────────────────────>
    >───── <statement-1> ; ─────────────────────────────────>
                .
                .
                .
    >───── <statement-n> ; ─────────────────────────────────>
    >───── END CASE; ───────────────────────────────────────┤
```

Syntax Semantics:

index
    This field can be any valid SDL/UPL identifier or expression that returns a binary value between 0 and n, inclusive and specifies the statement to be selected.

statement-0 through statement-n
    These fields can be any valid SDL/UPL statement and specify the statement to be performed.

Example 1:

```
CASE X;             % The value of X determines which procedure is
       PROC_A;      % performed.  X can vary in value from 0 through
       PROC_B;      % 2.  If the value of X is greater than the number
       PROC_C;      % of statements in the CASE statement, a run-time
     END CASE;      % error occurs.
```

Example 2:

```
CASE (A * B) MOD 2;             % The value of the expression is
    DO;                         % used to determine which statement
       IF X > 15 THEN UNDO;     % to perform.  A DO statement or
       X := X + 5;              % CASE statement is considered one
       END;                     % statement.
    CASE X;
       PROC_0;
       PROC_1;
         .
         .
         .
       PROC_20;
       END CASE;
END CASE;
```

Example Program:

```
DECLARE   NUMBER FIXED;

NUMBER := 0;
DO FOREVER;
   CASE NUMBER;
       DISPLAY "MARY";    % NUMBER = C
       DISPLAY "HAD";     % NUMBER = 1
       DISPLAY "A";       % NUMBER = 2
       DISPLAY "LITTLE";  % NUMBER = 3
       DISPLAY "LAMB";    % NUMBER = 4
   END CASE;
   IF (BUMP NUMBER) > 4 THEN UNDO;
END;

STOP;

FINI;

% This example program uses the CASE statement to
% display "MARY HAD A LITTLE LAMB" on the ODT
% and goes to end of job.  Each word is displayed
% on a separate line.
```

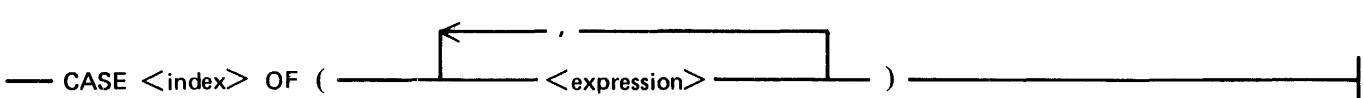
Output from Example Program:

```
CASEO =2037 BOJ. PP=4, MP=4 TIME = 11:57:32.4
% CASEO =2037 MARY
% CASEO =2037 HAD
% CASEO =2037 A
% CASEO =2037 LITTLE
% CASEO =2037 LAMB
CASEO =2037 EOJ. TIME = 11:57:38.2
```

CASE (format-2)

The CASE statement (format-2) uses the value of <index> to determine which expression to evaluate in the list of expressions contained in the parenthesis "( )" characters. The range of <index> is from 0 to n-1, where n is the number of expressions in the list.

SDL and UPL Syntax:



Syntax Semantics:

index
This field can be any valid SDL/UPL identifier or expression that returns a binary value between 0 and n−1, where n is the total number of expressions within the parenthesis "( )" characters and specifies the expression to be selected.

expression
This field can be any valid SDL/UPL number, identifier, or expression that returns a value and specifies the value. If selected by <index>, it is returned as a result of evaluating the CASE expression.

Example:

```
DECLARE  (A, B, C, F, I, J, Q) FIXED;          % Identifier A is
I := 2;                                        % assigned the value
J := 3;                                        % (A*B) + (A+B) MOD B.
    CASE J OF (Q*F-6, 9, 34+B, (A+6) MOD B, C);
```

Example Program:

```
DECLARE   NUMBER FIXED;

NUMBER := 0;
DO FOREVER;
   DISPLAY (CASE NUMBER OF ("MARY", "HAD", "A", "LITTLE", "LAMB"));
   IF (BUMP NUMBER) > 4 THEN UNDO;
END;

STOP;
FINI;

% This example program uses the CASE statement (format-2) to
% display "MARY HAD A LITTLE LAMB" on the ODT and goes
% to end of job.  Each word is displayed on a separate
% line.
```

Output from Example Program:

```
CASE0 =2037 BOJ. PP=4, MP=4 TIME = 11:57:32.4
% CASE0 =2037 MARY
% CASE0 =2037 HAD
% CASE0 =2037 A
% CASE0 =2037 LITTLE
% CASE0 =2037 LAMB
CASE0 =2037 EOJ. TIME = 11:57:38.2
```

ASSIGNMENT STATEMENT

The assignment statement is the only data-movement statement in SDL/UPL. Truncation and padding are performed across the assignment operator (:=). They are dependent upon the data type and length attributes of the data item as specified in the declaration statements. For data items with a CHARACTER data type, truncation of characters and padding of blank characters is on the right. For data items with a BIT or FIXED data type, truncation of data and padding of zeros is on the left.

Examples:

```
X := 0;        % Identifier X is assigned the value 0.

X := A;        % Identifier X is assigned the value of
               % identifier A.
```

NULL STATEMENT

The null statement performs a no-operation function during program execution. Two adjacent semicolon (;) characters are used to delimit a null statement.

The null statement is considered a complete statement that can be specified whenever the syntax requires a complete statement. Its most common usage is in the CASE and IF verbs to fulfill the syntax requirements and not to perform operations. The null statement can be specified in the READ, WRITE, and SPACE verbs.

The null statement can be specified to control events within a compound IF verb. However, this control is more readily accomplished if DO-groups are used within the compound IF verb.

SDL and UPL Syntax:

```
—— ; ——————————————————————————————————————————|
```

Example:

```
CASE DECODE;                % The identifier DECODE is used to select one
     PROC_A;     % 0        % of six statements within the CASE statement
     PROC_B;     % 1        % body.  If the value of identifier DECODE is
     ;           % 2        % a 2 or a 3, no operation is performed.
     ;           % 3
     PROC_C;     % 4
     PROC_D;     % 5
END CASE;
```

SECTION 9
VERBS

FORMAT OF THE VERB DESCRIPTION

All verbs that can be used in an SDL/UPL program are described in this section. Each verb is described separately. The SDL and UPL verb description is presented first, followed by the railroad syntax diagrams, the syntax semantics, examples, and an example program.

The valid constructs for the SDL compiler are presented in the SDL railroad syntax diagrams. The valid constructs for the UPL compiler are presented in the UPL railroad syntax diagrams, only if the UPL syntax is different from the SDL syntax. The description, syntax semantics, and examples show the action taken by the SDL and UPL compilers. Care must be taken to distinguish the differences between the two compilers when referencing the syntax semantics and examples.

ACCEPT

The ACCEPT verb causes the program to be suspended and to wait for input from the Operator Display Terminal (ODT). The input is provided to the program by way of the MCP AX input command which is entered by the system operator at the ODT.

The ODT input message is stored left-justified into <destination>. If the ODT input message is larger than <destination>, the message is truncated on the right. If the message is smaller, the message is padded on the right with blanks.

The actual input/output (I/O) operation processes the message as character data, regardless of the declared type of <destination>.

When the ACCEPT verb is performed, the MCP suspends the SDL/UPL program and sends the following message to the ODT. The (<usercode>) portion is optional.

(<usercode>) <program name> = <job number> ACCEPT

The following format is required to enter a message on the B 1000 computer system ODT.

<job number>AX <text> <ETX character>

The maximum length for the ODT input message is 69 characters.

SDL and UPL Syntax:

```
── ACCEPT <destination>; ──────────────────────────────────────┤
```

Syntax Semantics:

destination
This field can be any valid SDL/UPL identifier or an expression that generates an address.

Example Program:

```
DECLARE  MESSAGE  CHARACTER (69);

   DO FOREVER;

      ACCEPT MESSAGE;

      IF MESSAGE = "BYE" THEN UNDO;

      DISPLAY MESSAGE;

   END;

STOP;

FINI;

% This example program accepts a message from
% the ODT.  When a message is input, the program
% displays the message back onto the ODT.  If
% BYE is entered, the program goes to end of job.
```

ACCESS_FILE_INFORMATION

The ACCESS_FILE_INFORMATION verb causes the end-of-file pointer and the device type in the File Information Block (FIB) to be stored in <destination>. This information reflects the current status of the file in the program. The end-of-file pointer is the relative record number of the last record in the file. The device type is an MCP-maintained value that represents the hardware type of the file. For example, a device type of 16 represents a device type equal to DISK_PACK. Refer to the CHANGE verb in this section for a complete description of all the valid device types and associated device type codes.

The end-of-file pointer and the device type can be stored in BIT or CHARACTER data type format.

The following is the format for <destination> of data type BIT.

```
01 DESTINATION_VARIABLE BIT (30),
   03 EOF_POINTER         BIT(24),
   03 DEVICE_TYPE         BIT (6);
```

The following is the format for <destination> of data type CHARACTER.

```
01 DESTINATION_VARIABLE CHARACTER (10),
   03 EOF_POINTER         CHARACTER(8),
   03 DEVICE_TYPE         CHARACTER(2);
```

<file-identifier> must name a declared file. The return-type indicator (BIT or CHARACTER) must match the declared type of the variable. The information is returned to the address specified by <destination>. The format of the returned information varies with the return-type indicator. The file being accessed must be open to ensure that the File Information Block (FIB) exists.

SDL and UPL Syntax:

```
── ACCESS_FILE_INFORMATION  ( <file-identifier>, ──────────┬──── BIT ──────────┬──────>
                                                           └──── CHARACTER ────┘
>──── , <destination> ) ; ──────────────────────────────────────────────────────────┤
```

Syntax Semantics:

BIT
: The keyword BIT specifies that the data type of <destination> is equal to BIT.

CHARACTER
: The keyword CHARACTER specifies that the data type of <destination> is equal to CHARACTER.

destination
: This field can be any valid SDL identifier.

ACCESS_FILE_INFORMATION

The following summarizes the format of <destination> in the ACCESS_FILE_INFORMATION verb.

Item	BIT	CHARACTER
EOF_Pointer	24	8
Device type	6	2

file-identifier

This field is the name of the file to be interrogated. This file must be open prior to performing the ACCESS_FILE_INFORMATION verb.

Example Program:

```
FILE
   DISKFILE  (DEVICE = DISK SERIAL,
               RECORDS = 1/180,
               OPEN_OPTION = OUTPUT/NEW);

DECLARE
   01 DESTINATION_VARIABLE  CHARACTER (10),
      03 EOF_POINTER        CHARACTER (8),
      03 DEVICE_TYPE        CHARACTER (2),
   DATA                     CHARACTER (1);

DATA := "1";
WRITE DISKFILE (DATA);
ACCESS_FILE_INFORMATION (DISKFILE, CHARACTER, DESTINATION_VARIABLE);
DISPLAY "EOF POINTER = " CAT EOF_POINTER CAT " AND DEVICE TYPE IS "
        CAT DEVICE_TYPE;
CLOSE DISKFILE;
FINI;

% This example program writes one record to a disk file
% and obtains the end-of-file pointer and device type
% by using the ACCESS_FILE_INFORMATION verb.  The program
% subsequently displays the end-of-file pointer and
% device type on the system ODT, closes the disk file, and
% goes to end of job.
```

Output from Example Program:

```
%  TEST =6331 EOF POINTER = 00000001 AND DEVICE TYPE IS 15
```

BASE_REGISTER

The BASE_REGISTER verb returns a 24-bit value that is the current and absolute main-memory address of the beginning data space for the program.

In a multiprogramming environment, performing two separate BASE_REGISTER verbs can yield different results. Different results occur because the MCP can move the program to a new location in memory as memory space is required.

SDL Syntax:

```
—— BASE_REGISTER ——————————————————————————————————————————————————————————|
```

Example:

```
DECLARE  BASE  BIT (24);        % Identifier BASE contains the current
BASE := BASE_REGISTER;          % memory address of the program.
```

Example Program:

```
DECLARE  NEW_BASE_ADDRESS  BIT (24),
         SAVE_BASE_ADDRESS BIT (24);

SAVE_BASE_ADDRESS := BASE_REGISTER;

DISPLAY ("THE CURRENT BASE ADDRESS IS EQUAL TO " CAT
          CONVERT (SAVE_BASE_ADDRESS, CHARACTER));
DISPLAY ("ENTER ANY INPUT TO GO TO EOJ");
DO FOREVER;

   NEW_BASE_ADDRESS := BASE_REGISTER;

   IF (SAVE_BASE_ADDRESS /= NEW_BASE_ADDRESS)
   THEN DISPLAY ("THE BASE ADDRESS HAS CHANGED, THE NEW ADDRESS IS "
                 CAT CONVERT (BASE_REGISTER, CHARACTER));
   IF WAIT (TIME_TENTHS (5), SPO_INPUT_PRESENT)
   THEN STOP;
END;
FINI;

% This example program uses the BASE_REGISTER verb to display
% the current memory address of the beginning of the program,
% and then goes into a loop to check for a change in the base
% address.  If the address changes, the new address is displayed
% on the ODT.  If any message is accepted to the program, the
% program goes to end of job.
```

BINARY

The BINARY verb returns a FIXED data-type value which is the binary representation of the character string. Only the rightmost eight characters of the string are converted.

If the result of a BINARY verb returns a binary value greater than 24 bits (a decimal number greater than 16,777,215), the leftmost bits are truncated.

If the decimal number is greater than 8,388,607 ([2 exp 23] − 1), the returned value is a negative value because the leftmost bit is 1.

SDL and UPL Syntax:

```
—— BINARY ( <character-string> ) ——————————————————————————————|
```

Syntax Semantics:

character-string
: This field can be any valid group of characters that contain decimal digits and specifies the value to be converted.

Examples:

```
DECLARE CHAR     CHARACTER (7),
        RESULT   FIXED;

CHAR := "1234567";
RESULT := BINARY (CHAR);          % RESULT equals +1234567
```

BINARY

Example Program:

```
DECLARE
          RESULT          FIXED,
          ADDEND_ONE      CHARACTER (3),
          ADDEND_TWO      CHARACTER (3);

DO FOREVER;

  DISPLAY "ENTER ANY THREE DIGIT NUMBER, LEADING ZEROS ARE REQUIRED,";
  DISPLAY "OR ""BYE"" TO GO TO END-OF-JOB.";
  ACCEPT ADDEND_ONE;
  IF ADDEND_ONE = "BYE" THEN UNDO;
  DISPLAY "ENTER ANY THREE DIGITS FOR THE SECOND NUMBER, LEADING";
  DISPLAY "ZEROS ARE REQUIRED.";
  ACCEPT ADDEND_TWO;
  IF ADDEND_TWO = "BYE" THEN UNDO;
  RESULT := BINARY (ADDEND_ONE) + BINARY (ADDEND_TWO);
  DISPLAY "THE TOTAL EQUALS " CAT CONVERT (RESULT, CHARACTER, 4);

END;

STOP;

FINI;

% This example program accepts two numbers in character format
% from the ODT, uses the BINARY verb to add two numbers together,
% and displays the result on the ODT.  If BYE is entered, the
% program goes to end of job.
```

BINARY_SEARCH

The BINARY_SEARCH verb searches an ordered list of items that start at <start-record> for <number-of-records>. The occurrence number of the entry that matches is returned. If there is no match, an occurrence number equal to the entry immediately after the last entry in the list is returned.

SDL Syntax:

```
—— BINARY_SEARCH ( <start-record>, <compare-field>, <compare-value>, ————————>
>———— <number-of-records> ) ——————————————————————————————————————————————————|
```

Syntax Semantics:

start-record
: This field can be any valid SDL identifier or expression that returns a value and specifies the first structure with which to begin the search.

compare-field
: This field is a template which gives the relative offset and size in the structure of the 24-bit field that is being compared with <compare-value>. A template is an identifier whose address is relative to the beginning of a structure rather than base relative. A field in a structure declared REMAPS BASE has such an address.

compare-value
: This field is the value that is compared with <compare-field>. <Compare-value> is considered "on the left" in the compare relation.

number-of-records
: This field can be any valid SDL number, identifier, or expression that returns a binary value and specifies the total number of records to search for.

BINARY_SEARCH

Example Program:

```
RECORD   TABLE
         DATA   FIXED,
         KEY    FIXED;

DECLARE ODT_INPUT       CHARACTER (4),
        COUNT           FIXED,
        RESULT          FIXED,
        COMPARE_VALUE   FIXED,
        T (1024)        TABLE;

COUNT := 0;
DO BUILD_LINKS FOREVER;
   IF COUNT = 1024 THEN UNDO BUILD_LINKS;
   T(COUNT).KEY := COUNT;
   T(COUNT).DATA := (TIME (COUNTER, BIT) MOD 1024);
   BUMP COUNT;
END BUILD_LINKS;

DO FOREVER;
   DISPLAY ("ENTER ANY NUMBER FROM 0 TO 1023 OR ENTER BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE"
      THEN DO;
             DISPLAY ("GOOD BYE");
             STOP;
           END;
   COMPARE_VALUE := CONVERT (ODT_INPUT, FIXED);
   IF COMPARE_VALUE > 1023
      THEN DISPLAY (ODT_INPUT CAT " IS TOO LARGE");
      ELSE IF COMPARE_VALUE < 0
              THEN DISPLAY (ODT_INPUT CAT " IS TOO SMALL");

              ELSE DO;
                      RESULT := BINARY_SEARCH (T(0), KEY[0],
                                   COMPARE_VALUE, 1024);
                      IF RESULT = COMPARE_VALUE
                      THEN DISPLAY ("THE VALUE OF DATA FOUND IS " CAT
                               CONVERT (DATA [RESULT], CHARACTER));
                      ELSE DISPLAY ("SEARCH FAILED");
                   END;
END;
FINI;

% This example program shows one way to use the BINARY_SEARCH verb.
% The program first builds a table.  The operator is then requested
% to enter any number between 0 and 1023.  Using the accepted value
% the program searches through the table for an equal condition and
% if found displays the base relative address of the beginning of the
% table entry that it found.  If the search fails, the program displays
% SEARCH FAILED.   If BYE is entered, the program goes to end of job.
```

BUMP

The BUMP verb increments <identifier> by <increment-amount>. If the BY keyword is not specified, <identifier> is incremented by 1. If the BUMP verb is used in an expression, a descriptor of the identifier is placed on the evaluation stack.

If either <identifier> or <increment-amount> has a length greater than 24 bits, only the rightmost 24 bits are evaluated. If either <identifier> or <increment-amount> has a length less than 24 bits, <identifier> or <increment-amount> is padded with leading zeros. Character strings are treated as bit strings.

SDL and UPL Syntax:

```
── BUMP <identifier> ──┬─────────────────────────────┬──────────┤
                       └──── BY <increment-amount> ───┘
```

Syntax Semantics:

identifier
: This identifier can be any valid SDL/UPL identifier and specifies the field to be incremented.

increment-amount
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a 24-bit binary number and specifies the amount to increment <identifier>.

BY
: The keyword BY specifies that <increment-amount> follows.

Examples:

```
BUMP X;                        % Add 1 to X.

BUMP X BY 4;                   % Add 4 to X.

BUMP X BY Z;                   % Add the value of Z to X.

A := BUMP X BY Z;              % Add the value of Z to X, assign
                               % the sum to X, and assign the value
                               % of X to A.

IF (BUMP X BY Z) EQL ZERO      % Add the value of Z to X and store
   THEN ... ;                  % in X, and then perform the comparison.
   ELSE ... ;

BUMP A BY B := C;              % Assign the value of C to B and
                               % then add the value of C to A.
                               % Notice that the value of C is added to
                               % A because of the replacement delete
                               % left part operator.

X := BUMP A BY B := C;         % Replace B by the value of C, delete
                               % B, add the value of C to A, and assign
                               % the value to A and to X.
```

BUMP

```
PROC_B (BUMP X);            % Identifier X is incremented by 1
                            % and X is passed to procedure PROC_B.

PROC_B ((BUMP X));          % Identifier X is incremented by 1
                            % and the value of X is passed to
                            % procedure PROC_B.  The extra set of
                            % parentheses causes the value to be
                            % passed to PROC_B instead of the name
                            % X.
```

Example Program:

```
DECLARE   NUMBER  FIXED;

NUMBER := 0;

DO FOREVER;
   IF (BUMP NUMBER) > 10 THEN UNDO;
   DISPLAY CONVERT (NUMBER, CHARACTER);
END;

STOP;

FINI;

% This example program uses the BUMP verb to increment
% a number by one.  The resulting value of the number is
% displayed on the ODT.  The program increments and displays
% the number ten times and goes to end of job.
```

Output from the Example Program:

```
% BUMPO =6501 +0000001
% BUMPO =6501 +0000002
% BUMPO =6501 +0000003
% BUMPO =6501 +0000004
% BUMPO =6501 +0000005
% BUMPO =6501 +0000006
% BUMPO =6501 +0000007
% BUMPO =6501 +0000008
% BUMPO =6501 +0000009
% BUMPO =6501 +0000010
```

CHANGE

The CHANGE verb causes the SDL/UPL program to dynamically modify the attributes of a file during the execution of a program. The CHANGE verb must be specified after the file is declared. The change does not become effective until the file is opened. If the file to be changed is opened when the CHANGE verb is performed, the change is not effective until the file is closed and reopened.

Only those file attributes listed in the CHANGE verb are modified. Those omitted remain as previously set.

To effectively modify the attributes of a file, use the following procedure.

1. Close the file with a file attribute which causes the memory space for the File Information Block (FIB) to be released. If the memory space for the FIB is not released, the MCP does not rebuild the FIB, and any attempt to change the file attribute is disallowed. The following examples show four ways to close a file so that the memory space for the FIB is released.

   ```
   CLOSE FILE_A WITH LOCK;
   CLOSE FILE_B WITH RELEASE;
   CLOSE FILE_C WITH CRUNCH;
   CLOSE FILE_D WITH PURGE;
   ```

2. Modify the desired file attributes using the CHANGE verb.
3. Open the file explicitly by using the OPEN verb or implicitly by using the READ or WRITE verbs.

Refer to Table 9-1 for a complete description of the file attributes that can be specified with the CHANGE verb.

SDL and UPL Syntax:

```
                                  <---------- , ----------------
                                  |                            |
-- CHANGE <file-identifier> TO ( -------- <attribute> : = <value> --------- ) ; --|
```

Syntax Semantics:

file-identifier
: This file identifier can be any valid SDL/UPL file identifier and specifies the file to be modified.

attribute
: This field can be any valid file attribute and specifies the file attribute to be modified. Refer to FILE in Section 4 of this manual for a complete list of the valid file attributes.

value
: This field can be any valid SDL/UPL number, identifier, or expression that returns a value and specifies the file attribute value.

Table 9-1 shows all the valid values for each file attribute.

CHANGE

Table 9-1. Valid File Attribute Values

File Attribute	Value	Description
ALL_AREAS_AT_OPEN	0 1	Resets the attribute. Sets the attribute.
AREA_BY_CYLINDER	0 1	Resets the attribute. Sets the attribute.
BLOCKS_PER_AREA	n	Specifies the blocks per area for the file.
BUFFERS	<number-of-buffers>	Specifies the number of buffers.
DEVICE	<hardware variant> CAT <hardware type>	Refer to Table 8-2 for a complete list of the hardware variants and hardware types.
END_OF_PAGE_ACTION	0 1	Resets end-of-page reporting. Sets end-of-page reporting.
EU_DRIVE	<drive-number>	Specifies the disk drive number. EU_SPECIAL and EU_INCREMENTED must be set.
EU_INCREMENT	<drive-number>	Specifies the disk drive number. EU_SPECIAL and EU_INCREMENTED must be set.
EU_INCREMENTED	0 1	Resets EU_INCREMENTED. Sets EU_INCREMENTED.
EU_SPECIAL	0 1	Resets EU_SPECIAL. Sets EU_SPECIAL.
FILE_ID	"<file-identifier>"	Specifies the file identifier for the file.
FILE_TYPE	0 or 9 7 8 12	Specifies DATA file type. Specifies INTERPRETER file type. Specifies CODE file type. Specifies INTRINSIC file type.

CHANGE

Table 9-1. Valid File Attribute Values (Cont)

File Attribute	Value	Description
INVALID_CHARACTERS	0	Reports all lines containing invalid characters.
	1	Reports all lines containing invalid characters and stops the program.
	2	Reports once, that the file contains invalid characters.
	3	Does not report that the file contains invalid characters.
LABEL_TYPE	0	Use ANSI standard label.
	1	File is unlabeled.
	2	Use Burroughs standard (ANSI) label.
LOCK	0	Resets LOCK.
	1	Sets LOCK.
MULTI_FILE_ID	"<multi-file-id>"	Specifies the multifile identifier for the file.
MULTI_PACK	0	Places file on single disk pack.
	1	Places file on multiple disk packs.
NUMBER_OF_AREAS	n	Specifies the number of disk areas.
NUMBER_OF_STATIONS	n	Specifies the maximum number of stations for the remote file. The value of n can range from 0 to 999.
OPEN_ON_BEHALF_OF	0	Resets the OPEN_ON_BEHALF_OF boolean.
	1	Sets the OPEN_ON_BEHALF_OF boolean.

CHANGE

Table 9-1. Valid File Attribute Values (Cont)

File Attribute	Value	Description
OPEN_OPTION	12-bit field	Bit 0 – INPUT Bit 1 – OUTPUT Bit 2 – NEW Bit 3 – PUNCH Bit 4 – PRINT Bit 5 – NO_REWIND, INTERPRET Bit 6 – REVERSE, STACKERS Bit 7 – LOCK Bit 8 – LOCK_OUT
OPTIONAL	0	File must be present.
	1	File is optional.
PACK_ID	"<pack-identifier>"	Specifies the disk pack identifier.
PARITY	0	Specifies odd parity checking.
	1	Specifies even parity checking.
QUEUE_FAMILY_SIZE	n	Specifies the number of subqueues in the queue file.
QUEUE_MAX_MESSAGES	n	Specifies the maximum number of messages that the file can contain.
REMOTE_HEADERS	0	Resets the headers boolean for remote files.
	1	Sets the headers boolean for remote files.
RECORDS_PER_BLOCK	n	Specifies the number of records per block for the file.
RECORD_SIZE	n	Specifies the number of bytes per record.
REEL	n	Specifies the reel number.
REMOTE_KEY	0	Remote key is present on all read and write operations on the file.
	1	Remote key is not present.
SAVE	n	Specifies the number of days to save the file.

CHANGE

Table 9-1. Valid File Attribute Values (Cont)

File Attribute	Value	Description
SERIAL	6-character string	Specifies the tape serial number.
TRANSLATE	0	Resets translate.
	1	Sets translate.
TRANSLATE_FILE	"<file-identifier>"	Specifies the name of the translate table file identifier.
TRANSLATION	@(1)000@	Specifies EBCDIC translation.
	@(1)001@	Specifies ASCII translation.
	@(1)010@	Specifies BCL translation.
USE_INPUT_BLOCKING	0	Takes attributes from file declaration.
	1	Takes attributes from disk file header.
VARIABLE	0	File contains only fixed-length records.
	1	File contains variable-length records.
WORK_FILE	0	Does not insert job number in file identifier.
	1	Inserts job number in file identifier.

CHANGE

Table 9-2 shows the hardware code and variant for each hardware device type. If the device-type name has an asterisk (*) character on the left, the name is not a valid spelling for use with the CHANGE verb. The value is a 10-bit value where the leftmost four bits are the variant and the rightmost six bits are the hardware code.

Table 9-2. Valid DEVICE Type Values

Device Type Name	Hardware Code (bits 4-9)	Variant (bits 0-3)
* DATA RECORDER (80 column)	01	
CARD_PUNCH	02	(Same as PRINTER)
CARD_PUNCH FORMS	02	(Same as PRINTER FORMS)
PUNCH	02	(Same as PRINTER)
PUNCH FORMS	02	(Same as PRINTER FORMS)
* FDC 1	04	
READER_PUNCH PRINTER	05	(Same as PRINTER)
READER_PUNCH_PRINTER FORMS	05	(Same as PRINTER FORMS)
PUNCH_PRINTER	05	(Same as PRINTER)
PUNCH_PRINTER FORMS	05	(Same as PRINTER FORMS)
PAPER_TAPE_READER	06	
PAPER TAPE READER 1	07	
PRINTER	08	0 – BACKUP TAPE or DISK 1 – BACKUP TAPE 2 – BACKUP DISK 3 – BACKUP TAPE or DISK 4 – HARDWARE ONLY 5 – BACKUP TAPE ONLY 6 – BACKUP DISK ONLY 7 – BACKUP TAPE or DISK only
PRINTER FORMS	08	8 + (PRINTER Variant)
READER SORTER 2	09	
SORTER_READER	10	
READER_SORTER	10	
DISK_FILE (any head per track disk)	11	
DISK_FILE (1A, 1C, system-memory head per track disk)	12	(Same as DISK)
DISK (disk cartridge control 2 or 3)	13	(Same as DISK)
DISK (disk cartridge control 1)	14	(Same as DISK)
DISK_PACK (any 225, 205, or 206 disk pack)	15	(Same as DISK
DISK_PACK	16	(Same as DISK)
DISK (any disk)	17	0 -- Serial 1 -- Random
* 5-N DISK	18	(Same as DISK
CARD_READER (96 column)	19	

Table 9-2. Valid DEVICE Type Values (Cont)

Device Type Name	Hardware Code (bits 4-9)	Variant (bits 0-3)
PAPER_TAPE_PUNCH	20	(Same as PRINTER)
PAPER_TAPE_PUNCH FORMS	20	(Same as PRINTER FORMS)
CARD_READER (80 column)	21	
CARD_READER	21	
* SPO (supervisory printout)	22	
* ODT (operator display terminal)	23	
TAPE_NRZ (any 9-track nonreturn-to-zero, tape unit)	24	
TAPE_7 (any 7-track upright, tape unit)	25	
TAPE_PE (any 9-track phase-encoded, tape unit)	26	
TAPE (any tape unit)	27	
TAPE_9 (any 9-track tape unit)	28	
CASSETTE	30	
PRINTER (printer control 5)	31	(Same as PRINTER)
PRINTER (printer control 5)	31	(Same as PRINTER FORMS)
DISK_PACK (206 and 207 disk pack)	32	(Same as DISK)
PRINTER (printer control 7)	33	(Same as PRINTER)
PRINTER (printer control 7	33	(Same as PRINTER FORMS)
PORT	60	
QUEUE	61	
* QUEUE FILE OLD	62	
REMOTE	63	

Examples:

```
CHANGE MY_FILE TO (FILE_ID := "YOUR_FILE");

CHANGE LINE TO (LABEL_TYPE := 2, END_OF_PAGE_ACTION := 1);

CHANGE DISK_FILE TO (USE_INPUT_BLOCKING := 1, FILE_TYPE := 0);
```

CHANGE

Example Program:

```
FILE WORKFILE (DEVICE = DISK, LABEL = "MASTER"/"OLD");
ZIP "SO OPEN";
OPEN WORKFILE WITH NEW;
CLOSE WORKFILE WITH RELEASE;
CHANGE WORKFILE TO (FILE_ID := "NEW",
                    MULTI_FILE_ID := "MASTER");
OPEN WORKFILE WITH NEW;
CLOSE WORKFILE WITH RELEASE;
ZIP "RO OPEN";
STOP;
FINI;

% The example program shows one way to change the name of a file.
% The program sets the MCP OPEN option, opens the file, closes the
% file, changes the external file-id of the file, reopens the
% file, closes the file, resets the MCP OPEN option, and goes to
% end of job. The OPEN option is set in order to see the name of
% the file as it is opened by the MCP.
```

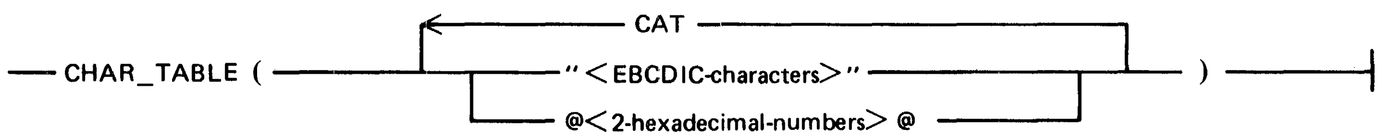
CHAR_TABLE

The CHAR_TABLE verb builds a 256-bit table string that describes a set-membership table, in which every member of the set is specified in the table string. Non-graphic characters are denoted in their hexadecimal (EBCDIC) form by concatenating bit strings into the table string. The table string generated by the CHAR_TABLE verb is a constant string that is built at compile time. Identifiers and expressions cannot be specified as elements of this table string.

The value of each character in the table string is used as its index into the table string. When a character is a member of the set described by the table string, its corresponding bit in the table string is set to @(1)1@. Position in the table string is based on the standard EBCDIC collating sequence.

The CHAR_TABLE verb is frequently used in conjunction with the REDUCE verb.

SDL and UPL Syntax:



Syntax Semantics:

EBCDIC-character
: This field can contain one or more EBCDIC characters and specifies the character(s) to be included as member(s) of the table.

2-hexadecimal-number
: The two digits that comprise a hexadecimal number are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. This number specifies the hexadecimal number to be included as a member of the table.

Example:

```
DECLARE  X    BIT (256);
X := CHAR_TABLE ("ABC" CAT @FF@ CAT "123");

% X is a 256-bit string where positions A, B, C, @FF@, 1,
% 2, and 3 are set to @(1)1@ and all other bit positions
% are set to @(1)0@.
```

CHAR_TABLE

Example Program:

```
DECLARE
        VOWEL_TABLE         BIT (256),
        STRING              REFERENCE,
        ODT_INPUT           CHARACTER (69),
        EOS_FLAG            BIT (1);

VOWEL_TABLE := CHAR_TABLE ("AEIOUaeiou");
DO FOREVER;
  DISPLAY ("ENTER CHARACTERS OR ENTER BLANK TO GO TO END-OF-JOB");
  ACCEPT ODT_INPUT;
  REFER STRING TO ODT_INPUT;
  REDUCE STRING UNTIL FIRST /= " ";
         ON EOS STOP;
  REDUCE STRING UNTIL FIRST IN VOWEL_TABLE;
         ON EOS DO;
                   DISPLAY ("NO VOWELS IN YOUR INPUT");
                   EOS_FLAG := @(1)1@;
                END;
  IF NOT EOS_FLAG
     THEN DISPLAY ("THE FIRST VOWEL IS " CAT SUBSTR(STRING,0,1));
  EOS_FLAG := @(1)0@;
END;

FINI;

% This example program accepts input from the ODT and displays
% the first, English-language vowel encountered in the characters
% that are accepted.  Entering a blank input message sends the
% program to end of job.
```

CHARACTER_FILL

The CHARACTER_FILL verb causes the leftmost eight bits of the source field to be written throughout the destination field.

SDL and UPL Syntax:

```
── CHARACTER_FILL (<destination>,<source>); ──────────────────────────┤
```

Syntax Semantics:

destination

This field can be any valid SDL/UPL identifier and specifies the name of the destination field. Array elements, records, structures, and simple identifiers are valid destination fields for <destination>.

source

This field can be any valid SDL/UPL literal, identifier, or expression that returns a value and specifies the value to be filled into <destination>. Only the leftmost eight bits (one character) of <source> are used.

Examples:

```
DECLARE
   ARRAY(10)              CHARACTER (5),
   FIELD                  CHARACTER (1);

RECORD
   FILL_RECORD
        CHAR_FIELD        CHARACTER (1),
        FIXED_FIELD       FIXED,
        BIT_FIELD_24      BIT (24),
        BIT_FIELD_10      BIT (10);

CHARACTER_FILL (ARRAY(5), " ");        % Fills element 5 of array
                                       % identifier ARRAY with blank
                                       % characters.

CHARACTER_FILL (FIELD, @00@);          % Fills FIELD with hexadecimal
                                       % value equal to @00@.

CHARACTER_FILL (FILL_RECORD, "A");     % Fills FILL_RECORD with
                                       % the character A.
```

CHARACTER_FILL

Example Program:

```
DECLARE
        ACCEPT_FIELD     CHARACTER (72),
        DISPLAY_FIELD    CHARACTER (72);

DO FOREVER;
   DISPLAY ("ENTER FILL CHARACTER OR BYE TO GO TO END OF JOB");
   ACCEPT ACCEPT_FIELD;
   IF ACCEPT_FIELD = "BYE" THEN UNDO;
   CHARACTER_FILL (DISPLAY_FIELD, ACCEPT_FIELD);
   DISPLAY (DISPLAY_FIELD);
END;

STOP;

FINI;

% This example program accepts characters from the ODT.  If BYE
% is entered, the program goes to end of job.  The program uses
% the CHARACTER_FILL verb to fill the DISPLAY_FIELD field.
% The DISPLAY_FIELD field is then displayed on the ODT.
```

CLEAR

The CLEAR verb moves zeros (0) to the array if the array is declared with a data type equal to BIT or FIXED. It also moves blanks to the array identifier if the array is declared with a CHARACTER data type.

The CLEAR verb is not valid for paged arrays.

SDL and UPL Syntax:

```
                 ┌←──────── , ─────────┐
── CLEAR ────────┴── <array-identifier> ──┴── ; ─────────────────┤
```

Syntax Semantics:

array-identifier
  This identifier can be any valid SDL/UPL array identifier and specifies the array to be cleared.

Example 1:

```
DECLARE  TABLE (10)  CHARACTER;
CLEAR TABLE;                          % Moves blank characters to the
                                      % array labeled TABLE.
```

Example 2:

```
DECLARE  TABLE (10)        CHARACTER,
         WORK_ARRAY (20)   FIXED;
CLEAR TABLE, WORK_ARRAY;                % Moves blank characters to the
                                        % array labeled TABLE and moves
                                        % zeros to the array labeled
                                        % WORK_ARRAY.
```

CLEAR

Example Program:

```
DECLARE   CHAR_ARRAY (2)    CHARACTER (1),
          FIXED_ARRAY (2)   FIXED;

CHAR_ARRAY (0) := "A";
CHAR_ARRAY (1) := "B";
DISPLAY ("THE CONTENTS OF CHAR_ARRAY BEFORE CLEAR ARE " CAT @7F@
          CAT CHAR_ARRAY (0) CAT @7F@ CAT " AND " CAT @7F@ CAT
          CHAR_ARRAY (1) CAT @7F@);
FIXED_ARRAY (0) := 111111;
FIXED_ARRAY (1) := 222222;
DISPLAY ("THE CONTENTS OF FIXED_ARRAY BEFORE CLEAR ARE " CAT @7F@
          CAT CONVERT (FIXED_ARRAY (0), CHARACTER)  CAT @7F@ CAT
          " AND " CAT @7F@ CAT CONVERT (FIXED_ARRAY (1), CHARACTER)
          CAT @7F@);

CLEAR CHAR_ARRAY, FIXED_ARRAY;

DISPLAY ("THE CONTENTS OF CHAR_ARRAY AFTER CLEAR ARE " CAT @7F@
          CAT CHAR_ARRAY (0) CAT @7F@ CAT " AND " CAT @7F@ CAT
          CHAR_ARRAY (1) CAT @7F@);
DISPLAY ("THE CONTENTS OF FIXED_ARRAY AFTER CLEAR ARE " CAT @7F@
          CAT CONVERT (FIXED_ARRAY (0), CHARACTER)  CAT @7F@ CAT
          " AND " CAT @7F@ CAT CONVERT (FIXED_ARRAY (1), CHARACTER)
          CAT @7F@);
DISPLAY ("GOOD BYE");
STOP;
FINI;

% This example program uses the CLEAR verb to clear two arrays
% and displays the value of each array before and after the
% CLEAR verb is performed.
```

Output from Example Program:

```
CLEARO =6912 BOJ. PP=4, MP=4 TIME = 15:28:37.0
% CLEARO =6912 THE CONTENTS OF CHAR_ARRAY BEFORE CLEAR ARE "A"
     AND "B"
% CLEARO =6912 THE CONTENTS OF FIXED_ARRAY BEFORE CLEAR ARE "+
    0111111" AND "+0222222"
% CLEARO =6912 THE CONTENTS OF CHAR_ARRAY AFTER CLEAR ARE " "
    AND " "
% CLEARO =6912 THE CONTENTS OF FIXED_ARRAY AFTER CLEAR ARE "+0
    000000" AND "+0000000"
% CLEARO =6912 GOOD BYE
CLEARO =6912 EOJ. TIME = 15:28:57.2
```

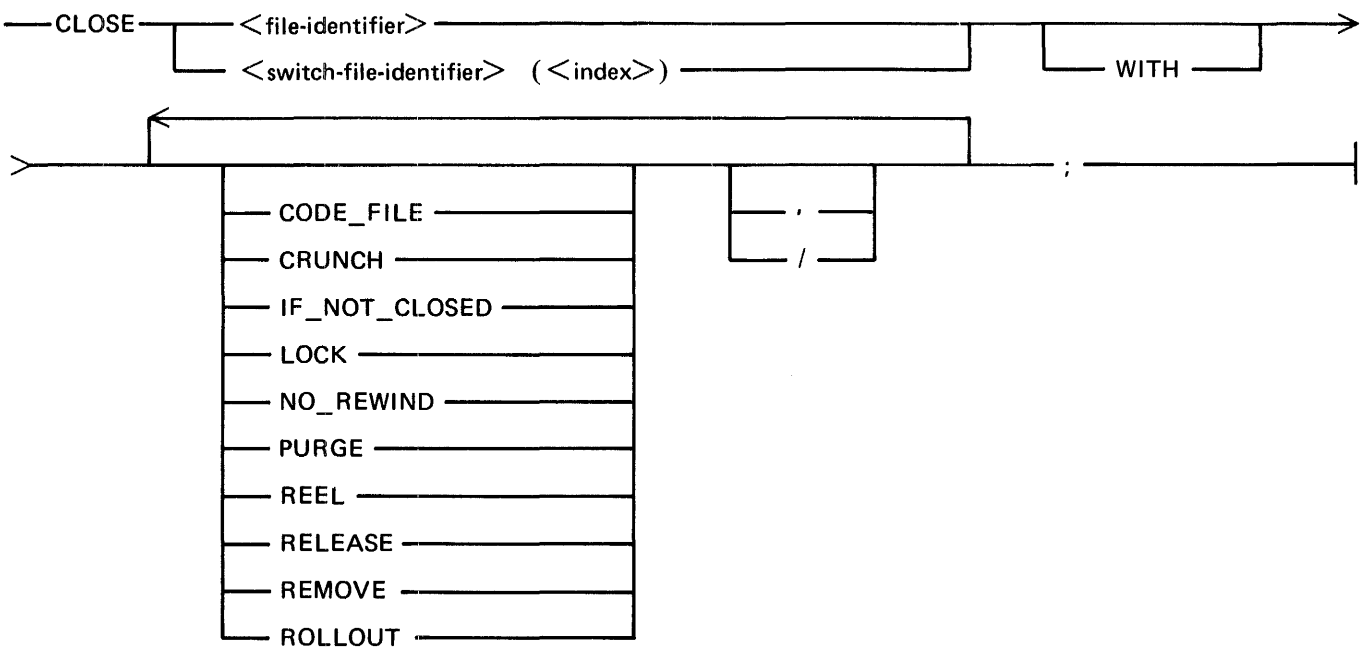
CLOSE

The CLOSE verb explicitly terminates program control over a file.

If there are no close attributes specified with the CLOSE verb, the program gives up control of the file to the MCP and the memory space is not released. If a read or write operation is attempted on the file, the file is reopened with the existing FIB. Even if an explicit open is done, the FIB is not rebuilt.

An implicit close is performed by the MCP when the program goes to end of job and when the file was not explicitly closed by the program. An implicit close with release is performed unless the attributes in the FILE declaration override the RELEASE close attribute.

SDL and UPL Syntax:



Syntax Semantics:

file-identifier
: This file identifier can be any valid SDL/UPL file identifier and specifies the file to be closed.

switch-file-identifier
: This file identifier can be any valid SDL/UPL switch file identifier and specifies the file to be closed.

WITH
: The keyword WITH is optional and specifies that close keyword options are to follow.

CODE_FILE
: The keyword CODE_FILE causes the SDL/UPL program to notify the MCP to close a file as a code file. A code file is a file that can be executed on the B 1000 computer system.

CLOSE

CRUNCH

The keyword CRUNCH causes the disk file header to be modified such that the AREAS file attribute is assigned a value of 1 and the BLOCKS PER AREA file attribute is assigned the actual size used. Also, the CRUNCH keyword causes the SDL/UPL program to notify the MCP to release all memory space used for the file and to enter the file name into the disk directory. The CRUNCH keyword applies only to disk files that are opened with the OUTPUT and NEW file attributes and to those that have only one area allocated.

IF_NOT_CLOSED

The keyword IF_NOT_CLOSED prevents the attempted close of an unopened file. The MCP terminates a program that attempts to close a file that is not open.

LOCK

The keyword LOCK causes the SDL/UPL program to notify the MCP to enter the file name into the disk directory and to release all memory space used for the file.

NO_REWIND

The keyword NO_REWIND causes the SDL/UPL program to notify the MCP to close a tape file without rewinding the tape.

PURGE

The keyword PURGE applies only to disk and tape files.

For disk files, PURGE causes the SDL/UPL program to notify the MCP to remove the file name from the disk directory, to release all memory space used for the file, and to return the disk space used by the file to the DISK.AVAILABLE table.

For tape files, PURGE causes the SDL/UPL program to notify the MCP to rewind and scratch the tape.

REEL

The keyword REEL causes the SDL/UPL program to notify the MCP to close the current reel of a multireel tape file and leave the actual file open.

RELEASE

The keyword RELEASE applies only to disk and tape files.

For disk files, the RELEASE keyword causes the SDL/UPL program to notify the MCP to release all the memory space used for the file and remove the file name from the disk directory. If the file is a new disk file, the RELEASE keyword does not lock the disk file in the disk directory. The LOCK keyword must be specified in order to lock a new disk file in the disk directory when the file is closed.

For tape files, the RELEASE keyword causes the SDL/UPL program to notify the MCP to rewind the tape and leave the tape in a ready state.

REMOVE

The keyword REMOVE causes the SDL/UPL program to notify the MCP to check the disk directory for a duplicate file name. If a duplicate file name is found, the MCP removes the old entry and updates the disk available table on the old file's disk pack.

ROLLOUT
The keyword ROLLOUT causes the SDL/UPL program to notify the MCP that the file is to be rolled out to disk.

,
The keysymbol comma (,) is optional and is used to separate the options of the CLOSE verb.

/
The keysymbol virgule (/) is optional and is used to separate the options of the CLOSE verb.

Examples:

```
CLOSE MASTERFILE;
CLOSE LINE RELEASE, IF_NOT_CLOSED;
CLOSE WORKFILE PURGE;
CLOSE TAPEFILE NO_REWIND;
CLOSE DISKFILE CRUNCH LOCK;
```

Example Program:

```
FILE LINE (DEVICE = PRINTER, RECORDS = 132/1),      % Declares the
     DISK (DEVICE = DISK, RECORDS = 180/20),        % files LINE, DISK,
     CARD (DEVICE = CARD_READER, RECORDS = 80/1), % CARD, and TAPE.
     TAPE (DEVICE = TAPE_PE, RECORDS = 180/1);

OPEN LINE WITH OUTPUT NEW;                            % Opens the files
OPEN DISK WITH OUTPUT NEW LOCK;                       % LINE, DISK, CARD,
OPEN CARD WITH INPUT;                                 % and TAPE.
OPEN TAPE WITH OUTPUT NEW;

ZIP "SO CLOS;";                                       % Sets the MCP CLOS
                                                      % option.

CLOSE LINE WITH RELEASE IF_NOT_CLOSED;                % Closes the files
CLOSE DISK WITH CRUNCH REMOVE;                        % LINE, DISK< CARD,
CLOSE CARD WITH RELEASE IF_NOT_CLOSED;                % and TAPE.
CLOSE TAPE WITH REEL;

ZIP "RO CLOS;";                                       % Resets the MCP
                                                      % CLOS option.
STOP;

FINI;

% This example program shows various ways to close files of
% different device types.  The MCP CLOS option is set to show
% how the MCP actually closes the file as a result of performing
% the CLOSE verb.
```

COMMUNICATE_WITH_GISMO

The COMMUNICATE_WITH_GISMO verb is used exclusively by the MCP, or by an SDL program that is to run without the MCP to communicate with GISMO. If an SDL program uses this verb while the MCP is running, the system halts with the L-register equal to @0D0040@ (A program other than the MCP attempted a COMMUNICATE_WITH_GISMO or GISMO_COMMUNICATE (T=LIMIT_REGISTER).

The value of <communicate> is made non-self-relative by pushing the value to the value stack, if necessary. The absolute address of <communicate> is stored into the T-register and its length is stored into the L-register. The appropriate swapper value is stored in the X-register and control is passed to GISMO. Any value returned by GISMO is described by the same descriptor on the evaluation stack that was used to pass a value to GISMO.

SDL Syntax:

```
—— COMMUNICATE_WITH_GISMO ( <communicate> ) ; ——————————————————————————————|
```

Syntax Semantics:

communicate
: This field can be any valid SDL literal, identifier or expression and specifies the information to be passed to GISMO.

Example:

```
DECLARE GISMO_INFO   BIT (24);
GISMO_INFO := 0;

COMMUNICATE_WITH_GISMO (@44@ CAT @111111@);

STOP;
FINI;

% This example performs the COMMUNICATE_WITH_GISMO
% verb to pass @44111111@ to GISMO.
```

COMMUNICATE

The COMMUNICATE verb passes control to the MCP. The information stored in <MCP-communicate> is given to the MCP to act upon.

SDL Syntax:

```
—— COMMUNICATE ( <MCP-communicate> ) ; ——————————————————————————————————————————|
```

Syntax Semantics:

MCP-communicate
: This field can be any valid SDL literal, identifier, or expression that returns a value and it must specify a valid MCP communicate.

COMPILE_CARD_INFO

The COMPILE_CARD_INFO verb stores the information used to initiate the compilation of this program into <destination>.

The following is the format of the information that is stored in <destination>.

Item	Data Type	Length
OBJECT NAME	CHARACTER	30
EXECUTE TYPE	CHARACTER	2
COMPILER PACK IDENTIFIER	CHARACTER	10
COMPILER INTERPRETER NAME	CHARACTER	30
COMPILER INTRINSIC NAME	CHARACTER	10
COMPILER PRIORITY	CHARACTER	2
COMPILER SESSION NUMBER	CHARACTER	6
COMPILER JOB NUMBER	CHARACTER	6
COMPILER 1ST AND 2ND NAMES	CHARACTER	20
COMPILER CHARGE NUMBER	CHARACTER	7
FILLER	CHARACTER	1
COMPILATION DATE AND TIME	BIT	36
FILLER	BIT	4
COMPILER USERCODE	CHARACTER	10
COMPILER PASSWORD	CHARACTER	10
COMPILER PARENT JOB NUMBER	CHARACTER	4
COMPILER PARENT QUEUE ID	CHARACTER	20
COMPILER_LS_BOOLEAN	CHARACTER	1
SECONDS_BEFORE_DECAY	CHARACTER	4
PRIVILEGED	CHARACTER	1
COMPILER_RESTRICTIONS	CHARACTER	2

SDL and UPL Syntax:

```
── COMPILE_CARD_INFO ( <destination> ) ; ──────────────────────────────┤
```

Syntax Semantics:

destination
This field can be any valid SDL/UPL identifier and specifies the data name in which to store the compile card information.

Example:

```
DECLARE COMPILER_INFORMATION CHARACTER (181); % Stores the compile
COMPILE_CARD_INFO (COMPILER_INFORMATION);     % card information
                                              % into identifier
                                              % COMPILER_INFORMATION.
```

COMPILE_CARD_INFO

Example Program:

```
DECLARE 01 CCI                             CHARACTER,
           03 OBJECT_NAME                  CHARACTER (30),
           03 EXECUTE_TYPE                 CHARACTER (2),
           03 COMPILER_PACK_ID             CHARACTER (10),
           03 COMPILER_INTERPRETER_NAME    CHARACTER (30),
           03 COMPILER_INTRINSIC_NAME      CHARACTER (10),
           03 COMPILER_PRIORITY            CHARACTER (2),
           03 COMPILER_SESSION_NUMBER      CHARACTER (6),
           03 COMPILER_JOB_NUMBER          CHARACTER (6),
           03 COMPILER_1ST_AND_2ND_NAMES   CHARACTER (20),
           03 COMPILER_CHARGE_NUMBER       CHARACTER (7),
           03 FILLER                       CHARACTER (1),
           03 COMPILATION_DATE_AND_TIME    BIT (36),
           03 FILLER                       BIT (4),
           03 COMPILER_USERCODE            CHARACTER (10),
           03 COMPILER_PASSWORD            CHARACTER (10),
           03 COMPILER_PARENT_JOB_NUMBER   CHARACTER (4),
           03 COMPILER_PARENT_QUEUE_ID     CHARACTER (20),
           03 COMPILER_LS_BOOLEAN          CHARACTER (1),
           03 SECONDS_BEFORE_DECAY         CHARACTER (4),
           03 COMPILER_PRIVILEGED          CHARACTER (1),
           03 COMPILER_RESTRICTIONS        CHARACTER (2);

COMPILE_CARD_INFO (CCI);

DISPLAY ("OBJECT NAME IS " CAT OBJECT_NAME);
DISPLAY ("EXECUTE TYPE IS " CAT EXECUTE_TYPE);
DISPLAY ("COMPILER PACK IDENTIFIER IS " CAT COMPILER_PACK_ID);
DISPLAY ("COMPILER INTERPRETER NAME IS " CAT
         COMPILER_INTERPRETER_NAME);
DISPLAY ("COMPILER INTRINSIC NAME IS " CAT COMPILER_INTRINSIC_NAME);
DISPLAY ("COMPILER PRIORITY IS " CAT COMPILER_PRIORITY);
DISPLAY ("COMPILER SESSION NUMBER IS " CAT COMPILER_SESSION_NUMBER);
DISPLAY ("COMPILER JOB NUMBER IS " CAT COMPILER_JOB_NUMBER);
DISPLAY ("COMPILER 1ST AND 2ND NAMES OF RUNNING PROGRAM IS " CAT
         COMPILER_1ST_AND_2ND_NAMES);
DISPLAY ("COMPILER CHARGE NUMBER IS " CAT COMPILER_CHARGE_NUMBER);
DISPLAY ("COMPILATION DATE AND TIME IS " CAT
         CONVERT (COMPILATION_DATE_AND_TIME, CHARACTER));
DISPLAY ("COMPILER USERCODE IS " CAT COMPILER_USERCODE);
DISPLAY ("COMPILER PASSWORD IS " CAT COMPILER_PASSWORD);
DISPLAY ("COMPILER PARENT JOB NUMBER IS " CAT
         COMPILER_PARENT_JOB_NUMBER);
DISPLAY ("COMPILER PARENT QUEUE IDENTIFIER IS " CAT
         COMPILER_PARENT_QUEUE_ID);
DISPLAY ("COMPILER LS BOOLEAN IS " CAT COMPILER_LS_BOOLEAN);
DISPLAY ("SECONDS BEFORE DECAY IS " CAT SECONDS_BEFORE_DECAY);
DISPLAY ("COMPILER PRIVILEGED IS " CAT COMPILER_PRIVILEGED);
DISPLAY ("COMPILER RESTRICTIONS IS " CAT COMPILER_RESTRICTIONS);
DISPLAY ("GOOD BYE");
STOP;
```

COMPILE_CARD_INFO

```
FINI;

% This example program uses the COMPILE_CARD_INFO verb and
% displays the information on the ODT.
```

Output from Example Program:

```
CO_CA_INFO =7102 BOJ. PP=4, MP=4 TIME = 16:30:46.2
% CO_CA_INFO =7102 OBJECT NAME IS                      CO_CA_INFO
% CO_CA_INFO =7102 EXECUTE TYPE IS 01
% CO_CA_INFO =7102 COMPILER PACK IDENTIFIER IS USER
% CO_CA_INFO =7102 COMPILER INTERPRETER NAME IS             SDL        INT
  ERP1M
% CO_CA_INFO =7102 COMPILER INTRINSIC NAME IS SDL.INTRIN
% CO_CA_INFO =7102 COMPILER PRIORITY IS 04
% CO_CA_INFO =7102 COMPILER SESSION NUMBER IS 000000
% CO_CA_INFO =7102 COMPILER JOB NUMBER IS 007102
% CO_CA_INFO =7102 COMPILER 1ST AND 2ND NAMES OF RUNNING PROGRAM IS
         CO_CA_INFO
% CO_CA_INFO =7102 COMPILER CHARGE NUMBER IS 0999999
% CO_CA_INFO =7102 COMPILATION DATE AND TIME IS 58508F4D1
% CO_CA_INFO =7102 COMPILER USERCODE IS
% CO_CA_INFO =7102 COMPILER PASSWORD IS
% CO_CA_INFO =7102 COMPILER PARENT JOB NUMBER IS 7000
% CO_CA_INFO =7102 COMPILER PARENT QUEUE IDENTIFIER IS SMCS      ##0000
  0005
% CO_CA_INFO =7102 COMPILER LS BOOLEAN IS 1
% CO_CA_INFO =7102 SECONDS BEFORE DECAY IS 0029
% CO_CA_INFO =7102 COMPILER PRIVILEGED IS 1
% CO_CA_INFO =7102 COMPILER RESTRICTIONS IS 00
% CO_CA_INFO =7102 GOOD BYE
CO_CA_INFO =7102 EOJ. TIME = 16:31:10.5
```

CONSOLE_SWITCHES

The CONSOLE_SWITCHES verb places a 24-bit, self-relative value of the 24 console switches on the top of the evaluation stack. This verb only applies to B 1720 computer systems.

SDL and UPL Syntax:

```
—— CONSOLE_SWITCHES ——————————————————————————————————————————|
```

Example:

```
DECLARE SWITCH_VALUES BIT (24);     % Identifier SWITCH_VALUES is
SWITCH_VALUES := CONSOLE_SWITCHES;  % assigned the current value of
                                    % the 24 console switches on the
                                    % B 1720 system.
```

Example Program:

```
DISPLAY ("THE CURRENT VALUE OF THE 24 CONSOLE SWITCHES EQUALS "
         CAT CONVERT (CONSOLE_SWITCHES, CHARACTER));
```

Output from Example Program:

```
SWITCHESO =5361 BOJ. PP=4, MP=4 TIME = 09:33:30.1
% SWITCHESO =5361 THE CURRENT VALUE OF THE 24 CONSOLE SWITCHES
   EQUALS AAAAAA
SWITCHESO =5361 EOJ. TIME = 09:33:35.2
```

CONTROL_STACK_BITS

The CONTROL_STACK_BITS verb leaves, on the top of the evaluation stack, a 24-bit, self-relative value with a BIT data type. The BIT data type is the number of bits left in the control stack until the control stack overflows.

SDL Syntax:

```
—— CONTROL_STACK_BITS ——————————————————————————————————————————|
```

Example:

```
DECLARE  BITS_LEFT  BIT (24);        % Assigns the identifier BITS_LEFT
BITS_LEFT := CONTROL_STACK_BITS;     % the number of bits left on the
                                     % control stack before overflow.
```

Example Program:

```
DISPLAY ("THE NUMBER OF BITS LEFT ON THE CONTROL STACK EQUALS "
        CAT CONVERT (CONTROL_STACK_BITS, CHARACTER));
```

Output from Example Program:

```
CONTROLO =5337 BOJ. PP=4, MP=4 TIME = 08:53:32.5
% CONTROLO =5337 THE NUMBER OF BITS LEFT ON THE CONTROL STACK
    EQUALS 002A0
CONTROLO =5337 EOJ. TIME = 08:53:36.7
```

CONTROL_STACK_TOP

The CONTROL_STACK_TOP verb returns a 24-bit value which is the base-relative address of the next entry to be placed on the control stack.

SDL Syntax:

```
—— CONTROL_STACK_TOP ————————————————————————————————————————————————————|
```

Example:

```
DECLARE TOP_OF_STACK_ADDR BIT (24);      % Identifier TOP_OF_STACK_ADDR
TOP_OF_STACK_ADDR := CONTROL_STACK_TOP;  % is assigned the value of the
                                         % next entry to be placed on
                                         % the control stack.
```

Example Program:

```
DISPLAY ("THE ADDRESS OF THE NEXT ENTRY TO BE PLACED ON THE CONTROL"
         CAT " EQUALS " CAT CONVERT (CONTROL_STACK_TOP, CHARACTER));
```

Output from Example Program:

```
CONTROLO =5349 BOJ. PP=4, MP=4 TIME = 09:12:25.2
% CONTROLO =5349 THE ADDRESS OF THE NEXT ENTRY TO BE PLACED ON THE
   CONTROL EQUALS 002880
CONTROLO =5349 EOJ. TIME = 09:12:30.5
```

CONVERT

The CONVERT verb causes <convert-value> to be changed from one data type to another. A data type keyword must be specified.

The keynumbers 1, 2, 3, and 4 are used only with bit-to-character or character-to-bit conversions. The keynumber specifies the number of bits in the bit string which correspond to a single character in the character string. The default keynumber is 4, which produces a hexadecimal conversion.

A bit-to-character conversion does not return decimal digits. To convert a bit string to decimal digits, store the bit string into a FIXED identifier, and then convert the FIXED identifier to a CHARACTER identifier. The DECIMAL verb can be used for the decimal conversions.

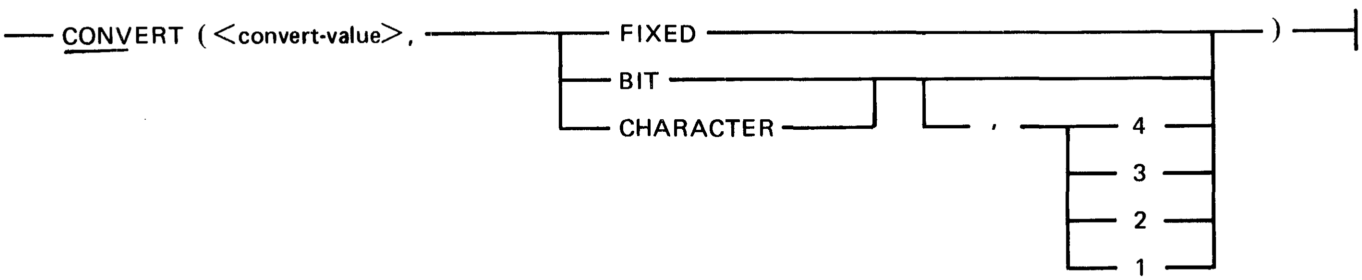
The conversion of data from type FIXED to type CHARACTER results in a sign and seven printable (EBCDIC) decimal numbers. The leading printable zeros and the arithmetic sign are not suppressed.

The following procedure must be performed to convert a field from data type CHARACTER to FIXED.

1. <convert-value> (with a CHARACTER data type) is scanned from left to right until a sign or non-space character is encountered. If the sign is negative, the FIXED number is expressed in the complement form of 2.
2. If a sign is encountered, it is noted and removed.
3. After encountering a sign or nonspace character, only the rightmost seven characters of <convert-value> are converted.
4. The rightmost four bits of each character are converted to a value between 0 and 15, inclusive. The leftmost four bits of each character are ignored. Each value is then multiplied by its respective ten's position and summed together. For example, the hexadecimal representation of the characters "AB5" is @(4)C1C2F5@. The rightmost four bits of each character is 125. The 2 is multiplied by 10, the 1 is multiplied by 100, and the sum of 5 + (2 * 10) + (5 * 100) is 525. The leftmost (sign) bit is ignored for decimal values in excess of +8,388,607 or −8,388,608.

CONVERT

SDL and UPL Syntax:



Syntax Semantics:

convert-value
: This field can be any valid SDL/UPL literal, identifier, or expression that returns an addressable item and specifies the value to be converted.

BIT
: The keyword BIT specifies that the resulting value of <convert-value> is to be a BIT data type.

CHARACTER
: The keyword CHARACTER specifies that the resulting value of <convert-value> is to be a CHARACTER data type.

FIXED
: The keyword FIXED specifies that the resulting value of <convert-value> is to be a FIXED data type.

1
: The keynumber 1 specifies the number of bits to be one and it is valid for character-to-bit and bit-to-character conversions.

2
: The keynumber 2 specifies the number of bits to be two and it is valid for character-to-bit and bit-to-character conversions.

3
: The keynumber 3 specifies the number of bits to be three and it is valid for character-to-bit and bit-to-character conversions.

4
: The keynumber 4 specifies the number of bits to be four and it is valid for character-to-bit and bit-to-character conversions.

CONVERT

Table 9-3 shows the possible data type conversion combinations.

Table 9-3. Data Type Conversion Combinations

Original Data Type	Data Type Desired	Result
BIT	BIT	No change.
BIT	CHARACTER	Bits are converted to characters based on bit group size. If no bit group size is specified, the bit group size defaults to 4.
BIT	FIXED	The rightmost 24 bits are returned to the expression.
CHARACTER	BIT	Characters are converted to bits based on bit group size. If no bit group size is specified, the bit group size defaults to 4.
CHARACTER	CHARACTER	No change.
CHARACTER	FIXED	The character expression is converted to a FIXED data type. The rightmost 4 bits of the 7 rightmost characters is converted to a binary number. If the minus sign character is the eighth character from the right, the 2's complement of the 24-bit field is returned.
FIXED	BIT	The data type is changed to BIT.
FIXED	CHARACTER	The numeric value of the expression is converted to decimal numbers in 8-bit EBCDIC character format. Leading zeros are not suppressed. The result is a CHARACTER data field of seven characters and a sign character.
FIXED	FIXED	No change.

Example 1:

```
CONVERT ("-72581", FIXED)         % The value -72581 is returned.
```

Example 2:

```
CONVERT (@(3)752@, CHARACTER, 4)  % The value "1EA" is returned.
```

Example 3:

```
CONVERT (@(1)11011@, FIXED)       % The value 27 is returned.
```

Example 4:

```
CONVERT ("132", BIT, 2)           % The value @(2)132@ is returned.
```

CONVERT

Example 5:

```
CONVERT ("132", BIT, 4)              % The value @(4)132@ is returned.
```

Example 6:

```
CONVERT ("2", BIT)                   % The value @(4)2@ is returned.
```

Example 7:

Assume that the identifier CX contains a character whose binary value is @(1)00001111@ and identifier B is declared as BIT (4).

```
B := CONVERT (CX, CHARACTER, 4);     % Identifier B is assigned the
                                     % hexadecimal value @F@ or
                                     % @(1)1111@.
```

Example 8:

Assume that the identifier CX contains a character whose binary value is @(1)00001111@ and identifier B is declared as BIT (4).

```
B := CONVERT (CX, CHARACTER, 3);     % Identifier B is assigned the
                                     % octal value of @(3)7@ or
                                     % @(1)111@. Only the rightmost
                                     % three bits of identifier CX are
                                     % assigned to B.
```

Example 9:

Assume identifier CARD contains the characters +4095 and FX is of data type FIXED.

```
FX := CONVERT (CARD, FIXED);         % Identifier FX is assigned the
                                     % hexadecimal value @000FFF@.
```

Example 10:

Assume identifier N is of data type FIXED with a value of +5 (000000000000000000000101) and identifier B is of data type BIT (8) with a value of @BC@ or @(1)10111100@.

```
OUTPUT := "ENTRY NO. "               % This statement assigns to the
   CAT CONVERT (N, CHARACTER)        % identifier OUTPUT the value of
   CAT " IS "                        % "ENTRY NO. +0000005 IS 2330".
   CAT CONVERT (B, CHARACTER, 2);
```

CONVERT

In example 10, the literal value "ENTRY$NO.$", the result of converting identifier N, the literal value "IS", and the result of converting identifier B, are made into a continuous string of data by using the CAT operator. The result of converting the FIXED value contained in identifier N to a printable character is +0000005, with no suppression of the 0's (zeros) or arithmetic sign. The result of converting the BIT value contained in identifier B, when using the character-to-quartal syntax as specified, is as follows:

```
10   11   11   00    (binary)
 2    3    3    0    (quartal)
F2   F3   F3   F0    (hexadecimal character)
```

Example Program:

```
DECLARE
    VALUE      CHARACTER (16),
    B          BIT (16),
    F          FIXED,
    I          FIXED,
    FLAG       BIT (1);

DO FOREVER;

    DISPLAY ("ENTER 16 1'S OR 0'S OR ENTER BYE TO GO TO EOJ");
    ACCEPT VALUE;
    IF VALUE = "BYE" THEN STOP;
    FLAG := @(1)0@;
    I := 0;
    DO LOOP FOREVER;
        IF ((SUBSTR(VALUE,I,1) = "1") OR (SUBSTR(VALUE,I,1) = "0"))
        THEN IF (BUMP I) > 15 THEN UNDO LOOP;
                                ELSE;
        ELSE DO;
                FLAG := @(1)1@;
                UNDO LOOP;
             END;
    END LOOP;
          IF FLAG = @(1)0@
          THEN DO;
                 B := CONVERT (VALUE, BIT, 1);
                 F := CONVERT (B, FIXED);
                 DISPLAY ("THE VALUE = " CAT (CONVERT (F, CHARACTER)));
               END;
          ELSE DISPLAY ("THE VALUE ENTERED WAS NOT ALL 1'S AND 0'S");
END;

FINI;

% This example program uses the CONVERT verb to calculate
% the decimal value of a 16-digit binary number.  The
% program accepts from the ODT a binary number with a data
% type of CHARACTER and converts this field to a field with
% data type of BIT.  The bit field is converted to a field
% with a data type of FIXED which is converted to a data
% type of CHARACTER and displayed on the ODT.
```

DATA__ADDRESS

The DATA__ADDRESS verb returns a 24-bit value that is the base-relative address of <identifier>.

SDL and UPL Syntax:

```
—— DATA_ADDRESS ( <identifier> ) ——————————————————————————————————————|
```

Syntax Semantics:

identifier
This identifier can be any valid SDL/UPL identifier and specifies the field name from which the address is to be determined.

Examples:

```
DECLARE
        BIT_FIELD           BIT (1),
        CHARACTER_FIELD     CHARACTER,
        FIXED_FIELD         FIXED,
        ADDRESS             BIT (24);

ADDRESS := DATA_ADDRESS (BIT_FIELD);     % ADDRESS is assigned the
                                         % address of BIT_FIELD.

ADDRESS := DATA_ADDRESS (CHAR_FIELD);    % ADDRESS is assigned the
                                         % address of CHAR_FIELD.

ADDRESS := DATA_ADDRESS (FIXED_FIELD);   % ADDRESS is assigned the
                                         % address of FIXED_FIELD.
```

Example Program:

```
DECLARE FIELD   BIT (1);
DISPLAY ("THE ADDRESS OF FIELD IS "
        CAT CONVERT (DATA_ADDRESS(FIELD),CHARACTER,4));
STOP;
FINI;

% This example program displays the base-relative address
% of identifier FIELD and goes to end of job.
```

DATA_LENGTH

The DATA_LENGTH verb returns the length of <data-item> in bits, regardless of the data type.

SDL Syntax:

── DATA_LENGTH (<data-item>) ────────────────────────────┤

Syntax Semantics:

data-item
This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the field in which to obtain the length.

Example:

```
LENGTH := DATA_LENGTH (A);      % Identifier LENGTH is assigned
                                % length of identifier A.
```

Example Program:

```
DECLARE  F     FIXED,
         C10   CHARACTER (10),
         B20   BIT (20);

DISPLAY "THE LENGTH OF IDENTIFIER F IS " CAT
          CONVERT (DATA_LENGTH (F), CHARACTER);
DISPLAY "THE LENGTH OF IDENTIFIER C10 IS " CAT
          CONVERT (DATA_LENGTH (C10), CHARACTER);
DISPLAY "THE LENGTH OF IDENTIFIER B20 IS " CAT
          CONVERT (DATA_LENGTH (B20), CHARACTER);
STOP;
FINI;

% This example program uses the DATA_LENGTH verb to find the
% length of fixed, character, and bit fields.
```

Output from Example Program:

```
D_LENGTHC =2145 BOJ. PF=4, MF=4 TIME = 15:30:36.9
% D_LENGTHC =2145 THE LENGTH OF IDENTIFIER F IS 000018
% D_LENGTHC =2145 THE LENGTH OF IDENTIFIER C10 IS 000050
% D_LENGTHC =2145 THE LENGTH OF IDENTIFIER B20 IS 000014
D_LENGTHC =2145 EOJ. TIME = 15:30:49.9
```

DATA__TYPE

The DATA__TYPE verb returns a bit string representing the data type of <data-item>. A value of @44@ represents a FIXED data field. A value of @48@ represents a CHARACTER data field. A value of @40@ represents a BIT data field.

SDL Syntax:

```
—— DATA_TYPE ( <data-item> ) ——————————————————————————————————————————————————|
```

Syntax Semantics:

data-item
: This field can be any valid SDL literal, identifier, or expression that returns a value and specifies data field in which to determine the data type.

Example:

```
TYPE := DATA_TYPE (A);   % Identifier TYPE is assigned the
                         % data type value of identifier A.
```

Example Program:

```
DECLARE  F     FIXED,
         C10   CHARACTER (10),
         B20   BIT (20);

DISPLAY "THE DATA TYPE OF IDENTIFIER F IS " CAT
          CONVERT (DATA_TYPE (F), CHARACTER);
DISPLAY "THE DATA TYPE OF IDENTIFIER C10 IS " CAT
          CONVERT (DATA_TYPE (C10), CHARACTER);
DISPLAY "THE DATA TYPE OF IDENTIFIER B20 IS " CAT
          CONVERT (DATA_TYPE (B20), CHARACTER);
STOP;
FINI;

% This example program displays the data type of fixed, character,
% and bit fields.
```

Output from Example Program:

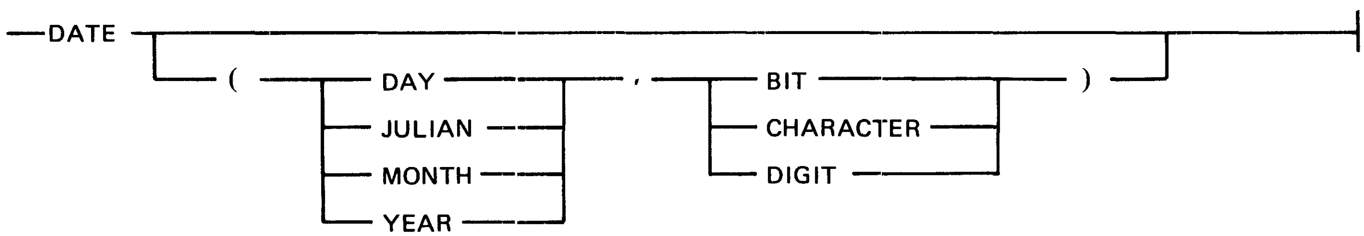
```
D_TYPEO =2150 BOJ. PP=4, MP=4 TIME = 15:35:27.6
% D_TYPEO =2150 THE DATA TYPE OF IDENTIFIER F IS 000044
% D_TYPEO =2150 THE DATA TYPE OF IDENTIFIER C10 IS 000048
% D_TYPEO =2150 THE DATA TYPE OF IDENTIFIER B20 IS 000040
D_TYPEO =2150 EOJ. TIME = 15:35:36.9
```

DATE

The DATE verb returns a bit or character string containing the current (run time) date.

Specifying DATE or DATE (MONTH, CHARACTER) returns the same result.

SDL and UPL Syntax:



Syntax Semantics:

DAY
: The keyword DAY causes the program to return the current day, month, year in the DDMMYY format, where DD is the day of the month, MM is the month, and YY is the year.

JULIAN
: The keyword JULIAN causes the program to return the current year and julian day in the YYDDD format, where YY is the year and DDD is the julian day of the year.

MONTH
: The keyword MONTH causes the program to return the current month, day, and year in the MMDDYY format, where MM is the month, DD is the day of the month, and YY is the year.

YEAR
: The keyword YEAR causes the program to return the current year, month, and day in the YYMMDD format, where YY is the year, MM is the month, and DD is the day of the month.

DATE

BIT

The keyword BIT causes the program to return the DAY, JULIAN, MONTH, and YEAR specifications in the following formats:

```
DAY           BIT (16),
          CC  BIT (5),
          MM  BIT (4),
          YY  BIT (7);

JULIAN        BIT (16),
          YY  BIT (7),
          DDD BIT (9);

MONTH         BIT (16),
          MM  BIT (4),
          CC  BIT (5),
          YY  BIT (7);

YEAR          BIT (16),
          YY  BIT (7),
          MM  BIT (4),
          CC  BIT (5);
```

DIGIT

The keyword DIGIT causes the program to return the DAY, JULIAN, MONTH, and YEAR specifications in the following formats:

```
DAY           BIT (24),
          CC  BIT (8),
          MM  BIT (8),
          YY  BIT (8);

JULIAN        BIT (20),
          YY  BIT (8),
          DDD BIT (12);

MONTH         BIT (24),
          MM  BIT (8),
          CC  BIT (8),
          YY  BIT (8);

YEAR          BIT (24),
          YY  BIT (8),
          MM  BIT (8),
          CC  BIT (8);
```

DATE

CHARACTER

The keyword CHARACTER causes the program to return the DAY, JULIAN, MONTH, and YEAR specifications in the following formats:

```
DAY            BIT (48),
          DD   BIT (16),
          MM   BIT (16),
          YY   BIT (16);

JULIAN         BIT (40),
          YY   BIT (16),
          DDD  BIT (24);

MONTH          BIT (48),
          MM   BIT (16),
          DD   BIT (16),
          YY   BIT (16);

YEAR           BIT (48),
          YY   BIT (16),
          MM   BIT (16),
          DD   BIT (16);
```

Table 9-4 shows the format and length of each option.

Table 9-4. Format and Length of each DATE Verb Option

Option	**Format**	**Bit Length**	**Digit Length**	**Character Length**
JULIAN	YY/DDD	7/9	2/3	2/3
MONTH	MM/DD/YY	4/5/7	2/2/2	2/2/2
DAY	DD/MM/YY	5/4/7	2/2/2	2/2/2
YEAR	YY/MM/DD	7/4/5	2/2/2	2/2/2

NOTES

YY represents the year, DD or DDD represents the day, and MM represents the month.

Digits are equal to four bits, which are two decimal digits per byte. Bytes are 8 bits long.

Characters are equal to eight bits or one byte.

DATE

Example:

```
DECLARE D BIT (24),
        J CHARACTER (40),
        M BIT (16),
        Y BIT (24);

D := DATE(DAY, DIGIT)
J := DATE(JULIAN, CHARACTER);
M := DATE(MONTH, BIT);
Y := DATE(YEAR, DIGIT);

% If the system's date is December 3, 1979, then variables D, J,
% M, and Y have the following bit and hexadecimal values:
%
% D = @(1)000000110000110001001111@
%     @(4)030A8F@
%
% J = @(1)1111011111111001111100111111100111110111@
%   = @(4)F7F9F3F3F7@
%
% M = @(1)1100000111001111@
%   = @(4)A1AF@
%
% Y = @(1)1001111110000011@
%   = @(4)9F83@
```

DATE

Example Program:

```
DECLARE
    01  DAY_MONTH_YEAR,
        03  D_DD          CHARACTER (2),
        03  D_MM          CHARACTER (2),
        03  D_YY          CHARACTER (2),

    01  JULIAN_DATE,
        03  J_YY          CHARACTER (2),
        03  J_DD          CHARACTER (3),

    01  MONTH_DAY_YEAR,
        03  M_MM          CHARACTER (2),
        03  M_DD          CHARACTER (2),
        03  M_YY          CHARACTER (2),

    01  YEAR_MONTH_DAY,
        03  Y_YY          CHARACTER (2),
        03  Y_MM          CHARACTER (2),
        03  Y_DD          CHARACTER (2);

DAY_MONTH_YEAR := DATE (DAY, CHARACTER);
MONTH_DAY_YEAR := DATE (MONTH, CHARACTER);
YEAR_MONTH_DAY := DATE (YEAR, CHARACTER);
JULIAN_DATE    := DATE (JULIAN, CHARACTER);

DISPLAY ("THE JULIAN DATE IS " CAT J_YY CAT "/" CAT J_DD);
DISPLAY ("THE DAY/MONTH/YEAR IS " CAT D_DD CAT "/" CAT D_MM
          CAT "/" CAT D_YY);
DISPLAY ("THE MONTH/DAY/YEAR IS " CAT M_MM CAT "/" CAT M_DD
          CAT "/" CAT M_YY);
DISPLAY ("THE YEAR/MONTH/DAY IS " CAT Y_YY CAT "/" CAT Y_MM
          CAT "/" CAT Y_DD);

STOP;
FINI;

% This example program displays the current date in
% the JULIAN, DAY, MONTH, YEAR formats on the ODT.
```

DC_INITIATE_IO

The DC_INITIATE_IO verb causes a data communications read or write operation for the port and channel address specified by <port> and <channel>, respectively. It also uses the input/output (I/O) descriptor address specified by <I/O-descriptor-address>.

SDL Syntax:

```
—— DC_INITIATE_IO ( <port>, <channel>, <I/O-descriptor-address> ) ; ——————————————|
```

Syntax Semantics:

port
: This field can be any valid SDL literal, identifier, or expression that returns a binary value and specifies the port on which the I/O operation is to occur.

channel
: This field can be any valid SDL literal, identifier, or expression that returns a binary value and specifies the channel on which the I/O operation is to occur.

I/O-descriptor-address
: This field can be any valid SDL literal, identifier, or expression that returns a 24-bit value and specifies the base-relative address of the I/O descriptor.

Example:

```
DECLARE PORT             BIT (4),              % The input/output,
        CHANNEL          BIT (4),              % defined by the I/O
        DESC_ADDRESS     BIT (24);             % descriptor at the
PORT := 2;                                     % address of identifier
CHANNEL := 0;                                  % DESC_ADDRESS, is
DESC_ADDRESS := @000F52@;                      % initiated.
DC_INITIATE_IO (PORT, CHANNEL, DESC_ADDRESS);
```

DEBLANK

The DEBLANK verb repeatedly increments the address field of the descriptor for <first-character> until <first-character> describes a non-blank character.

SDL Syntax:

── DEBLANK (<first-character>) ; ──────────────────────────┤

Syntax Semantics:

first-character
This field can be any simple SDL identifier and specifies the first character to be examined.

Example:

```
DECLARE  DATA        CHARACTER (20),           % The reference identifier
         REF_DATA    REFERENCE;                % REF_DATA contains the
DATA := "     ABCDEFGHIJKLMNO";                % first non-blank character
REFER REF_DATA TO SUBSTR (DATA, 0, 1);         % "A" after the DEBLANK verb
DEBLANK (REF_DATA);                            % is performed.
```

Example Program:

```
DECLARE  ODT_INPUT   CHARACTER (50),
         REFER_ODT   REFERENCE;

DO FOREVER;
   DISPLAY ("ENTER ANY 50 CHARACTERS OR ENTER B TO GO TO EOJ");
   ACCEPT ODT_INPUT;
   REFER REFER_ODT TO SUBSTR (ODT_INPUT, 0, 1);

   DEBLANK (REFER_ODT);

   IF REFER_ODT = "B" THEN DO;
                              DISPLAY ("GOOD BYE");
                              STOP;
                           END;
   DISPLAY ("THE FOLLOWING IS THE FIRST CHARACTER THAT IS NOT BLANK");
   DISPLAY (REFER_ODT);
END;
FINI;

% This example program accepts from the ODT any 50-character
% string and displays the first non-blank character in the
% string.  If B is entered, the program goes to end of job.
```

DECIMAL

The DECIMAL verb causes the value of <string> to be converted to a string of decimal digits. If the value generated has a length greater than 24 bits, only the rightmost 24 bits are converted.

The number of characters returned is controlled by the value <string-size>. A maximum of eight decimal digits can be returned, even if the value of <string-size> is greater than 8. If <string-size> specifies fewer character positions than the total number of decimal digits in <string>, the resulting decimal number is truncated on the left.

SDL and UPL Syntax:

```
—— DECIMAL (<string>,<string-size>) ——————————————————————————————————|
```

Syntax Semantics:

string
: This field can be any valid SDL/UPL literal, identifier, or expression that generates a CHARACTER data type and specifies the name of the field to be converted.

string-size
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a 24-bit binary value and specifies the number of characters in <string> to be converted to decimal digits. The range of value for <string-size> is from 1 to 8, inclusive.

Example 1:

```
NUMBER := DECIMAL ("12345",5);          % Converts all five characters of
                                        % the literal 12345 to the decimal
                                        % digits 12345 and assigns them to
                                        % identifier NUMBER.
```

Example 2:

```
NUMBER := DECIMAL (FIELD_A,8);          % Converts eight of the characters
                                        % in FIELD_A to decimal digits
                                        % and assigns them to identifier
                                        % NUMBER.
```

Example 3:

```
NUMBER :=                               % Evaluates the expression,
 DECIMAL ((BUMP FIELD_B BY 3),8);       % converts eight of the characters
                                        % in the expression to decimal
                                        % digits, and assigns them to
                                        % NUMBER.
```

Example 4:

```
NUMBER := DECIMAL (A, 4);               % Identifier A is converted from
                                        % a 24-bit binary value to a
                                        % 4-character numeric string.
                                        % The value is assigned to
                                        % identifier NUMBER.
```

DECIMAL

Example 5:

```
NUMBER := DECIMAL (@FF@, 3);          % Identifier NUMBER is assigned
                                      % the value 255.
```

Example Program:

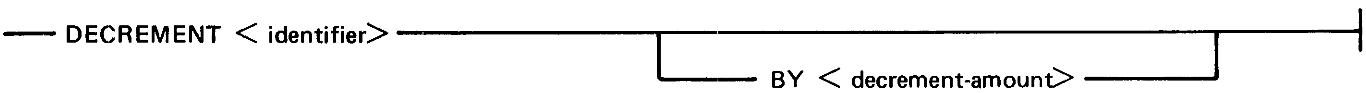
```
DECLARE  FIELD   CHARACTER (6);

DO FOREVER;
   DISPLAY ("ENTER ANY 6 CHARACTERS OR ENTER BYE TO GO TO EOJ");
   ACCEPT FIELD;
   IF FIELD = "BYE" THEN STOP;
   DISPLAY (DECIMAL (CONVERT (FIELD, BIT, 4), 8));
END;

FINI;

% This example program accepts a 6-character field from
% the ODT and displays its hexadecimal value using the
% DECIMAL verb.
```

DECREMENT

The DECREMENT verb decrements <identifier> by the amount specified by <decrement-amount>. If the BY keyword is not specified, <identifier> is decremented by 1. If the DECREMENT verb is used in an expression, a descriptor of <identifier> is placed on the evaluation stack.

If either <identifier> or <decrement-amount> has a length greater than 24 bits, only the rightmost 24 bits are evaluated. If either <identifier> or an expression has a length less than 24 bits, <identifier> or <decrement-amount> is padded with leading zeroes. Character strings are treated as bit strings.

SDL and UPL Syntax:



Syntax Semantics:

identifier
: This field can be any valid SDL/UPL identifier and specifies the name of the field to be decremented.

BY
: The keyword BY is required if <decrement-amount> is specified.

decrement-amount
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value and specifies the amount that is subtracted from <identifier>.

Examples:

```
DECREMENT X;                        % Subtract 1 from X.

DECREMENT X BY 4;                   % Subtract 4 from X.

DECREMENT X BY Z;                   % Subtract the value of Z from X.

A := DECREMENT X BY Z;              % Subtract the value of Z from X,
                                    % assign the value to X, and then
                                    % assign the value of X to A.

IF (DECREMENT X BY Z) EQL ZERO      % Subtract the value of Z from X,
    THEN ... ;                      % assign the value to X, and then
    ELSE ... ;                      % perform the comparison.

DECREMENT A BY B := C;              % Assign the value of C to B and
                                    % then subtract the value of C
                                    % from A. Notice that C is
                                    % subtracted from A because of
                                    % the replacement delete left
                                    % part operator.
```

DECREMENT

```
X := DECREMENT A BY B := C;        % Replace B by the value of C,
                                   % delete B, subtract C from A,
                                   % and assign the value to A and
                                   % to X.

PROC_B (DECREMENT X);              % Identifier X is decremented by 1
                                   % and then X is passed to procedure
                                   % PROC_B.

PROC_B ((DECREMENT X));            % Identifier X is decremented by 1
                                   % and then the value of X is passed
                                   % to procedure PROC_B.  The extra
                                   % set of parentheses causes the
                                   % value to be passed to PROC_B
                                   % instead of the name X.
```

Example Program:

```
DECLARE   NUMBER  FIXED;

NUMBER := 11;

DO FOREVER;
   IF (DECREMENT NUMBER) = 0 THEN STOP;
   DISPLAY CONVERT (NUMBER, CHARACTER);
END;

STOP;
FINI;

% This example program uses the DECREMENT verb to decrement
% a number by one and display the resulting value of the
% number.  The program decrements and displays the number
% ten times on the ODT and goes to end of job.
```

Output from the Example Program:

```
% DECREMENTO =6501 +0000010
% DECREMENTO =6501 +0000009
% DECREMENTO =6501 +0000008
% DECREMENTO =6501 +0000007
% DECREMENTO =6501 +0000006
% DECREMENTO =6501 +0000005
% DECREMENTO =6501 +0000004
% DECREMENTO =6501 +0000003
% DECREMENTO =6501 +0000002
% DECREMENTO =6501 +0000001
```

DELIMITED_TOKEN

The DELIMITED_TOKEN verb scans the identifier that has <first-character-address> as its first character until one of the two delimiters specified by <delimiter> is encountered. The remaining portion of the identifier that begins with <first-character-address> is stored in <result-reference-identifier>.

The delimiter characters used by the SDL compiler are the percent sign (%) and semicolon (;) characters.

SDL Syntax:

```
—— DELIMITED_TOKEN (<first-character-address>, <delimiters>, ——————————>
>———— <result-reference-identifier>) ——————————————————————————————————|
```

Syntax Semantics:

first-character-address
: This field can be any valid SDL identifier and specifies the address of the first character in the character string to be scanned.

delimiters
: This field can be a character or bit string with a length equal to 16 bits. Each 8-bit byte specifies one of two delimiter tokens.

result-reference-identifier
: This field can be any valid SDL reference identifier and specifies the name of the field in which to store the string of characters.

Example:

```
DECLARE FIRST_CHAR      REFERENCE,          % The identifier
        RESULT          REFERENCE,          % RESULT_STRING is
        CHAR_STRING     CHARACTER (15),     % assigned the value
        RESULT_STRING   CHARACTER (15);     % of "123456789".
CHAR_STRING := "123456789;ABCDE";
REFER FIRST_CHAR
 TO SUBSTR (CHAR_STRING, 0, 1);
RESULT_STRING :=
DELIMITED_TOKEN (FIRST_CHAR, ";%", RESULT);
```

DELIMITED_TOKEN

Example Program:

```
DECLARE  ODT_INPUT          CHARACTER (50),
         RESULT             REFERENCE,
         FIRST_CHARACTER    REFERENCE;

DO FOREVER;
   DISPLAY ("ENTER ANY 50-CHARACTERS TO BE SCANNED OR ENTER BYE FOR"
            CAT " EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                               DISPLAY ("GOOD BYE");
                               STOP;
                            END;

   REFER FIRST_CHARACTER TO SUBSTR (ODT_INPUT, 0, 1);
   DISPLAY ("THE DELIMITED CHARACTERS FOLLOW");

   DISPLAY (DELIMITED_TOKEN (FIRST_CHARACTER, " %", RESULT));

END;
FINI;

% This example program uses the DELIMITED_TOKEN verb to scan a
% character string that is accepted from the ODT.  The delimiter
% characters used are the blank character and the percent sign (%)
% character.  If BYE is entered, the program goes to end of job.
```

DESCRIPTOR

The DESCRIPTOR verb places a descriptor on the evaluation stack, which is the data descriptor of an identifier. The DESCRIPTOR verb can appear as the object of a replacement, thereby providing easy access to any part of a descriptor.

A descriptor contains the data type, length, and base-relative address of <simple-identifier> or <array-identifier>.

SDL Syntax:

```
── DESCRIPTOR  ( ──┬──── <simple-identifier> ────┬── ) ──────────────────┤
                   └──── <array-identifier> ─────┘
```

Syntax Semantics:

simple-identifier
: This field can be any valid SDL identifier and specifies the field name to obtain the data descriptor information.

array-identifier
: This field can be any valid SDL array identifier and specifies the array name needed to obtain the data descriptor information.

Examples:

```
SUBBIT (DESCRIPTOR (X), 4, 2 ) := 2;    % Assigns the value 2 to
                                        % the data type portion of
                                        % DESCRIPTOR (X).

DESCRIPTOR (X) := DESCRIPTOR (Y);       % Forces both identifiers X
                                        % and Y to describe the same
                                        % data name.  However, if X
                                        % and Y are not both simple
                                        % identifiers or arrays the
                                        % results are incorrect.
```

DISABLE_INTERRUPTS

The DISABLE_INTERRUPTS verb suppresses all interrupts until an ENABLE_INTERRUPTS verb is performed.

This verb is for MCP use only and cannot be used by a program when the MCP is running.

SDL Syntax:

── DISABLE_INTERRUPTS ; ──────────────────────────────────┤

Example:

```
DISABLE_INTERRUPTS;   % Causes all interrupts to be suppressed.
```

DISPATCH

The DISPATCH verb causes an input/output (I/O) operation to begin on the port and channel address specified by <port-and-channel>. It uses the I/O descriptor specified by <I/O-descriptor-address>. The DISPATCH verb is only used by the MCP or by a standalone SDL program that does not run with the MCP. If the DISPATCH verb is performed when the MCP is running, the MCP discontinues the program with the following program abort message:

INVALID OPERATOR

The DISPATCH verb returns one of the following three values.

Value	Description
0	Dispatch register lock bit is set
1	Successful dispatch
2	Successful dispatch, but device is missing

SDL Syntax:

```
—— DISPATCH ( <port-and-channel>, <I/O-descriptor-address> ) ——————————————|
```

Syntax Semantics:

port-and-channel
This field can be any valid SDL literal, identifier, or expression that returns a binary value and specifies the port and channel address for the I/O operation. The rightmost seven bits of <port-and-channel> are used. The leftmost three bits are the port number and the rightmost four bits are the channel number.

I/O-descriptor-address
This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the absolute address of the I/O descriptor. The rightmost 24 bits of <I/O-descriptor-address> are used.

DISPATCH

Example:

```
RECORD IO_DESC
       ACTUAL_END       BIT (24),
       RESULT_STATUS    BIT (24),
       OP               BIT (24),
       A_ADDRESS        BIT (24),
       B_ADDRESS        BIT (24),
       C_ADDRESS        BIT (24);

DECLARE  D              IO_DESC,
         RESULT         BIT (24),
         BUFFER         BIT (1440);

D.RESULT_STATUS := 0;
D.OP := @180000@;          % Read Operation
D.A_ADDRESS := DATA_ADDRESS (BUFFER);
D.B_ADDRESS := DATA_ADDRESS (BUFFER) + LENGTH (BUFFER);
D.C_ADDRESS := @070E41@;  % Sector Address

RESULT := DISPATCH (@(1)11110001@, DATA_ADDRESS (D.RESULT_STATUS));

% If RESULT = 0, then dispatch register lock bit is set.
% If RESULT = 1, then successful dispatch.
% If RESULT = 2, then successful dispatch, but missing device.
```

DISPLAY

The DISPLAY verb causes the SDL/UPL program to write a message to the Operator Display Terminal (ODT).

The following is the format of the output message that is written to the ODT. The (<usercode>) portion is optional.

% (<usercode>) <program-name> = <program number> <message text>

The displayed message is distinguished from the MCP-generated messages by the leading percent sign (%) character.

SDL and UPL Syntax:

```
── DISPLAY ( <display-identifier> ) ──────────────┬──────────────────┬─── ; ──────────┤
                                                  └── , CRUNCHED ────┘
```

Syntax Semantics:

display-identifier
    This field can be any valid SDL/UPL literal, identifier, or expression that returns an addressable value and specifies the value to be displayed on the ODT.

CRUNCHED
    The keyword CRUNCHED deletes trailing blanks and substitutes one blank for each occurrence of multiple embedded blanks.

Examples:

```
DISPLAY "HI THERE";                        % Displays on the ODT the
                                           % message "HI THERE".

DISPLAY ("PLEASE LOAD FORM " CAT           % Displays on the ODT the
         FORM_NUMBER), CRUNCHED;           % message "PLEASE LOAD
                                           % FORM " followed by the
                                           % value of FORM_NUMBER.

DISPLAY (MESSAGE);                         % Displays on the ODT the
                                           % value of MESSAGE.
```

DISPLAY

Example Program:

```
DECLARE  YOUR   CHARACTER (5),
         COMMA  CHARACTER (2),
         ROW    CHARACTER (4),
         BOAT   CHARACTER (5);

YOUR := " YOUR";
COMMA := ", ";
ROW := " ROW";
BOAT := " BOAT";

DISPLAY (ROW CAT COMMA CAT ROW CAT COMMA CAT ROW CAT YOUR CAT BOAT);
DISPLAY ("GENTLY DOWN THE STREAM");

STOP;
FINI;

% This example program uses the DISPLAY verb to display on the
% ODT the message "ROW, ROW, ROW YOUR BOAT GENTLY DOWN THE STREAM".
```

Output from Example Program:

```
DISPLAYO =2467 BOJ. PP=4, MP=4 TIME = 07:55:12.3
% DISPLAYO =2467  ROW,  ROW,  ROW YOUR BOAT
% DISPLAYO =2467 GENTLY DOWN THE STREAM
DISPLAYO =2467 EOJ. TIME = 07:55:17.3
```

DISPLAY_BASE

The DISPLAY_BASE verb stores, on the top of the evaluation stack, a 24-bit, self-relative value with a BIT data type that is the base-relative address of the base of the display stack.

SDL Syntax:

```
—— DISPLAY_BASE ————————————————————————————————————————————————————|
```

Example:

```
DECLARE  BASE_ADDRESS BIT (24);   % Identifier BASE_ADDRESS is assigned
BASE_ADDRESS := DISPLAY_BASE;     % the value of the base-relative
                                  % address of the display stack.
```

Example Program:

```
DISPLAY ("THE ADDRESS OF THE DISPLAY STACK EQUALS " CAT
        CONVERT (DISPLAY_BASE, CHARACTER));
STOP;
FINI;
```

Output from Example Program:

```
DISPLAYO =5535 BOJ. PP=4, MP=4 TIME = 15:17:54.2
% DISPLAYO =5535 THE ADDRESS OF THE DISPLAY STACK EQUALS 0027D0
DISPLAYO =5535 EOJ. TIME = 15:17:58.8
```

DUMP__FOR__ANALYSIS

The DUMP__FOR__ANALYSIS verb causes the MCP to create a file known as the dumpfile. This dumpfile reflects the status of the program at the point at which the DUMP__FOR__ANALYSIS verb is performed. After the dumpfile is created, program execution continues with the statement immediately following the DUMP__FOR__ANALYSIS verb. Refer to the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108966, for the syntax of the "PM" MCP command used to analyze and print the dump.

After the dumpfile is created, enter one of the following commands to execute the DUMP/ANALYZER program. The DUMP/ANALYZER program generates a printer listing that shows the status of the program at the time the DUMP__FOR__ANALYSIS verb was performed.

PM <dumpfile-id>; or EXECUTE DUMP/ANALYZER FILE DUMPFILE NAME <dumpfile-id>;

SDL and UPL Syntax:

```
—— DUMP_FOR_ANALYSIS; ——————————————————————————————————————|
```

Examples:

```
DUMP;

DUMP_FOR_ANALYSIS;
```

Example Program:

```
DISPLAY ("THIS PROGRAM CAUSES A DUMPFILE TO BE CREATED OF ITSELF");

DUMP_FOR_ANALYSIS;

STOP;
FINI;

% This example program displays "THIS PROGRAM CAUSES A DUMPFILE
% TO BE CREATED OF ITSELF" and goes to end of job.
```

Output from Example Program:

```
DUMPO =2640 BOJ. PP=4, MP=4 TIME = 15:28:40.1
% DUMPO =2640 THIS PROGRAM CAUSES A DUMPFILE TO BE CREATED OF ITSELF
DUMPO =2640 "DUMPFILE/1237"
DUMPO =2640 EOJ. TIME = 15:28:46.5
```

DYNAMIC__MEMORY__BASE

The DYNAMIC__MEMORY__BASE verb returns a 24-bit value that is the base-relative address in which the dynamic memory portion of the program begins.

SDL and UPL Syntax:

```
—— DYNAMIC_MEMORY_BASE ——————————————————————————————————————————————————————————————————|
```

Example:

```
DECLARE MEMORY   BIT(24);
MEMORY := DYNAMIC_MEMORY_BASE;          % The identifier MEMORY is
                                        % assigned the address of the
                                        % starting location of the
                                        % program's dynamic memory.
```

Example Program:

```
DISPLAY ("THE DYNAMIC MEMORY FOR THIS PROGRAM BEGINS AT "
        CAT CONVERT (DYNAMIC_MEMORY_BASE, CHARACTER));
STOP;
FINI;
```

Output from Example Program:

```
DYNAMICO =2660 BOJ. PP=4, MP=4 TIME = 16:18:22.5
% DYNAMICO =2660 THE DYNAMIC MEMORY FOR THIS PROGRAM BEGINS AT 003200
DYNAMICO =2660 EOJ. TIME = 16:18:24.9
```

ENABLE_INTERRUPTS

The ENABLE_INTERRUPTS verb causes the MCP to return to the normal interrupt-processing mode after a DISABLE_INTERRUPTS verb has been performed.

This verb is for MCP use only and a program cannot use this verb when the MCP is running.

SDL Syntax:

```
—— ENABLE_INTERRUPTS; ——————————————————————————————————————————————————|
```

Example:

```
ENABLE_INTERRUPTS;   % Causes the MCP to return to the normal
                     % interrupt-processing mode.
```

ENTER_COROUTINE

The ENTER_COROUTINE verb is used in conjunction with the EXIT_COROUTINE verb and causes the current code address to be placed on the program pointer stack. The number of entries specified in <coroutine-table> are placed onto the program pointer stack. The address of the next instruction is taken from the entry address specified in <coroutine-table>.

When the ENTER_COROUTINE verb is performed for the first time, <coroutine-table> must already be set up. This is accomplished by making the first executable statement in <coroutine-table> an EXIT_COROUTINE statement. The first entrance to the coroutine is then accomplished by a procedure call.

The ENTER_COROUTINE verb is not symmetric. The routine performing the ENTER_COROUTINE verb is a master to the slave routine performing the EXIT_COROUTINE verb.

SDL Syntax:

—— ENTER_COROUTINE (<coroutine-table>) ; ——————————————————————|

Syntax Semantics:

coroutine-table
This field can be any valid SDL table identifier and specifies a table with the following format.

```
01 COROUTINE_TABLE,
   03 NUMBER_OF_ENTRIES     BIT (4),
   03 ENTRY_ADDRESS         BIT (32),
   03 PPS_COPY              BIT (32);
```

Example:

```
DECLARE I        FIXED,
        TABLE    BIT(4+17*32);
PROCEDURE SLAVE;
  EXIT_COROUTINE (TABLE);        % Sets up table
  DO FOREVER;
     BUMP I BY 2;
     DISPLAY (DECIMAL (I, 6));
     EXIT_COROUTINE (TABLE);   % Resets table
  END;
END SLAVE;
PROCEDURE MASTER;
  SLAVE;                         % Call for table set up
  I := 0;
  DO FOREVER;
     BUMP I BY 3;
     DISPLAY (DECIMAL (I, 6));
     ENTER_COROUTINE (TABLE);  % Uses table
  END;
END MASTER;
```

ENTER_COROUTINE

The following is displayed if the example is performed.

Occurrence Number	Value of I Displayed
1	000003
2	000005
3	000008
4	000010
.	.
.	.
.	.
2n	5*n
2n +1	5*n +3
.	.
.	.
.	.

ERROR_COMMUNICATE

The ERROR_COMMUNICATE verb causes the value of <error-message> to be put on the evaluation stack as a descriptor. The MCP error communication is then performed, and the program is discontinued.

If the 6-bit identifier MCP_NUMBER is equal to 29, the MCP uses the 16-bit identifier MESSAGE_LENGTH as the length of the message and the 24-bit identifier MESSAGE_ADDRESS as the base-relative address of the program abort message to be displayed on the ODT. If the 6-bit field MCP_NUMBER is not equal to 29, the predefined MCP program abort message, represented by the MCP_NUMBER, is displayed on the ODT.

SDL Syntax:

```
ERROR_COMMUNICATE ( <error-message> ) ;
```

Syntax Semantics:

error-message
: This field can be any valid SDL identifier or expression that returns a value and specifies either a predefined MCP program abort message or a program-defined, program abort message.

The following is the format of <error-message>.

```
01 ERROR_MESSAGE,
   03 FILLER            BIT (2),
   03 MCP_NUMBER        BIT (6),
   03 MESSAGE_LENGTH    BIT (16),
   03 MESSAGE_ADDRESS   BIT (24);
```

The following are the predefined MCP program abort messages and their respective numbers.

Error Number	Program Abort Message
1	PROGRAM POINTER/EVALUATION STACK OVERFLOW
2	CONTROL STACK OVERFLOW
3	NAME/VALUE STACK OVERFLOW
4	REMAP AREA HAS INSUFFICIENT LENGTH
5	INVALID PARAMETER (passed to a procedure)
6	INVALID SUBSTRING (or SUBBIT)
7	INVALID SUBSCRIPT
8	INVALID RETURN (OF VALUE FROM PROCEDURE)
9	INVALID CASE
10	DIVIDE BY ZERO (could be in a MOD)
11	INVALID INDEX
12	MEMORY PARITY or READ OUT OF BOUNDS ON B1720
13	INVALID OPERATOR
14	INVALID PARAMETER TO VALUE DESCRIPTOR
15	CONVERT ERROR
16	STACK OVERFLOW
17	UNINITIALIZED DATA ITEM
18	ATTEMPTED TO WRITE OUT OF BOUNDS
19	EXPONENT OVERFLOW

ERROR_COMMUNICATE

Error Number	Program Abort Message
20	EXPONENT UNDERFLOW
21	EXPRESSION OUT OF RANGE
22	SUPERFLUOUS EXIT
23	OUT OF MEMORY SPACE
24	INVALID LINK
25	TYPE ERROR
26	INTEGER OVERFLOW
27	MESSAGE TRANSFER DATA AREA IS NOT PRESENT
28	MESSAGE TRANSFER INVALID DATA TEMPLATE
29	(user supplied message)
30	PARAMETER TO DYNAMIC DECLARATION OUT OF RANGE
31	INVALID TRANSLATE
32	INVALID SUBPROGRAM TYPE
33	REFERENCE ASSIGNMENT LENGTH MISMATCH

Example:

```
ERROR_COMMUNICATE (@0200000000000@);  % Causes the program abort
                                      % message CONTROL STACK
                                      % OVERFLOW to be displayed
                                      % on the ODT.
```

Example Program:

```
DECLARE ODT_INPUT  CHARACTER (50);
DISPLAY ("ENTER THE ERROR MESSAGE DESIRED OR ENTER BYE FOR EOJ");
ACCEPT ODT_INPUT;
IF ODT_INPUT = "BYE" THEN DO;
                             DISPLAY ("GOOD BYE");
                             STOP;
                          END;

ERROR_COMMUNICATE (@1D@ CAT @01902 CAT DATA_ADDRESS (ODT_INPUT));

STOP;
FINI

% This example program accepts the error message from the ODT and
% performs the ERROR_COMMUNICATE verb.  The error message is
% included in the terminate message displayed on the ODT by the
% MCP.  If BYE is entered, the program goes to end of job.
```

EVALUATION_STACK_TOP

The EVALUATION_STACK_TOP verb stores a 24-bit value on the top of the evaluation stack. This value is the base-relative address of the top of the evaluation stack before the verb is performed.

SDL Syntax:

```
—— EVALUATION_STACK_TOP ——————————————————————————————————————————————————————————|
```

Example:

```
DECLARE  TOP_OF_STACK  BIT (24);        % Identifier TOP_OF_STACK is
TOP_OF_STACK := EVALUATION_STACK_TOP;   % assigned the base address
                                        % of the top of the evaluation
                                        % stack.
```

Example Program:

```
DISPLAY ("THE ADDRESS OF THE TOP OF THE EVALUATION STACK EQUALS "
         CAT CONVERT (EVALUATION_STACK_TOP, CHARACTER));
STOP;
FINI;
```

Output from Example Program:

```
EVALUATED =5537 BOJ. PP=4, MP=4 TIME = 15:19:29.1
% EVALUATED =5537 THE ADDRESS OF THE TOP OF THE EVALUATION STACK
   EQUALS 002820
EVALUATED =5537 EOJ. TIME = 15:19:32.9
```

EXECUTE

The EXECUTE verb causes the operation specified in the operation-list to be performed by the SDL interpreter.

The EXECUTE verb is used only for the experimental design of new operation codes and results in the display of a BRANCH TO INVALID OP CODE program abort message on the ODT. The program is then discontinued.

SDL Syntax:

```
              ┌<──── , ─────────────┐
──EXECUTE ( ──┴──── <operation-list> ┴── ) ─────────────────────┤
```

Syntax Semantics:

operation-list
: This field can be any valid SDL identifier or expression. It specifies the operation code to be executed by the interpreter and the operands to be used by the interpreter.

Example:

```
DECLARE   A    FIXED,                   % Assigns identifier C
          B    FIXED,                   % the result of the AND
          C    BIT (24);                % logical operation that
                                        % is specified by the
C := EXECUTE (A, B, @(1)1111000001@);   % EXECUTE verb.

STOP;
FINI;
```

EXIT__COROUTINE

The EXIT__COROUTINE verb is used in conjunction with the ENTER__COROUTINE verb and causes the current nesting level to be stored in the number of entries specified in <coroutine-table>. The current code address is stored in the entry address specified in <coroutine-table>. The number of the entries that is specified in <coroutine-table>, on the top of the program pointer stack, is then copied to the program-pointer-stack-copy field (PPS__COPY) specified in <coroutine-table>. If the number of the entries is 0 (zero), then nothing is copied and an implicit UNDO statement is performed. The implicit UNDO statement uses the number of entries specified in <coroutine-table> as the number of entries on top of the program pointer stack.

The EXIT__COROUTINE verb can appear only within procedures that have no parameters and no local data, that is, those procedures which do not change the control stack.

SDL Syntax:

```
—— EXIT_COROUTINE (<coroutine-table>) ; ————————————————————————|
```

Syntax Semantics:

coroutine-table
: This field can be any valid SDL table identifier and specifies the table with the following format.

```
01 COROUTINE_TABLE,
   03 NUMBER_OF_ENTRIES     BIT (4),
   03 ENTRY_ADDRESS         BIT (32),
   03 PPS_COPY              BIT (32);
```

Example:

For an example of the EXIT__COROUTINE verb usage, refer to the ENTER__COROUTINE verb.

FETCH

The FETCH verb causes the result of an input/output (I/O) operation to be returned to the SDL program. If there is a high-priority interrupt, then that interrupt is stored in <result-descriptor-address>. If there is no high-priority interrupt and <I/O-reference-address> is non-zero, only an interrupt on an I/O descriptor with a reference address equal to <I/O-reference-address> is stored in <result-descriptor-address>. <I/O-reference-address> is stored in the leftmost 24 bits of <result-descriptor-address>. If there are no interrupts, then zeros are stored in <IO-reference-address> and <result-descriptor-address>.

The FETCH verb is for MCP use only or for an SDL program that is to run without the MCP.

SDL Syntax:

```
── FETCH ( <I/O-reference-address>, <port-and-channel-address>, ──────────>
>──────── <result-descriptor-address> ) ; ─────────────────────────────────┤
```

Syntax Semantics:

I/O-reference-address
: This field can be any valid SDL identifier or expression that returns a 24-bit value and specifies the reference address of the I/O operation.

port-and-channel-address
: This field can be any valid SDL literal, identifier, or expression that returns a 7-bit value and specifies the port and channel address. The first three bits specify the port address and the last four bits specify the channel address.

result-descriptor-address
: This field can be any valid SDL identifier and specifies the destination field in which to store the result descriptor address for a high-priority interrupt. This field is zero if there was no high-priority interrupt.

Example:

```
DECLARE  IO_REF_ADDR            BIT (24),
         PORT_CHANNEL_ADDR      BIT (7),
         RESULT_DESC_ADDR       BIT (24);

 IO_REF_ADDR := 0;
 PORT_CHANNEL_ADDR := @(1)010@ CAT @(1)0000@;

 FETCH    (IO_REF_ADDR, PORT_CHANNEL_ADDR, RESULT_DESC_ADDR);

 DISPLAY ("THE FOLLOWING RESULT DESCRIPTOR INFORMATION IS FOR PORT "
          CAT "2 AND CHANNEL 0");
 DISPLAY ("THE RESULT DESCRIPTOR ADDRESS IS " CAT
          CONVERT (RESULT_DESC_ADDR, CHARACTER));
 DISPLAY ("THE I/O REFERENCE ADDRESS IS " CAT
          CONVERT (IO_REF_ADDR, CHARACTER));
 STOP;
 FINI;
```

FETCH_COMMUNICATE_MSG_PTR

The FETCH_COMMUNICATE_MSG_PTR verb returns the RS_COMMUNICATE_MSG_PTR information if the RS_MCP_BIT field is set. Otherwise, the RS_REINSTATE_MSG_PTR information is returned.

SDL Syntax:

```
—— FETCH_COMMUNICATE_MSG_PTR ——————————————————————————————————————————|
```

Example:

```
DESCRIPTOR (COMM_MSG) :=
   VALUE_DESCRIPTOR (FETCH_COMMUNICATE_MSG_PTR);

% Identifier COMM_MSG describes the communicate message, that is
% assuming that the message was described by a non-self-relative
% descriptor.
```

FIND_DUPLICATE_CHARACTERS

The FIND_DUPLICATE_CHARACTERS verb scans <reference-identifier-1> for the first three or more contiguous characters that are identical. For example, the three characters AAA qualify as duplicate characters, while the two characters AA do not. The value of <reference-identifier-1> is modified if duplicate characters are encountered. The new value has the same character string except this character string begins immediately after the first duplicate character. The value of <count-identifier> is the number of duplicate characters found. The value of <character-identifier> is the duplicate character found. The value of <reference-identifier-2> is the original character string of <reference-identifier-1>, except this character string ends with the character immediately preceding the duplicate characters.

The FIND_DUPLICATE_CHARACTERS verb is helpful in a data communications environment where it can be used to compact messages, especially when blank characters are common.

SDL and UPL Syntax:

```
—— FIND_DUPLICATE_CHARACTERS (<reference-identifier-1>, ——————————>
>—————— <count-identifier>, <character-identifier>, ——————————————>
>—————— <reference-identifier-2>) ; ——————————————————————————————|
```

Syntax Semantics:

reference-identifier-1
: This field can be any valid SDL/UPL reference identifier and specifies the character string that is to be scanned. The value of this identifier is modified when the FIND_DUPLICATE_CHARACTERS verb is performed. The new value of <reference-identifier-1> is a character string that begins with the first character immediately following the duplicate characters that are found.

count-identifier
: This field can be any valid SDL/UPL identifier with a FIXED data type. After the FIND_DUPLICATE_CHARACTER verb is performed, the value contained in <count-identifier> is the number of duplicate characters found. For example, if the value equaled +0000007, the FIND_DUPLICATE_CHARACTERS verb found seven duplicate characters in the character string.

character-identifier
: This field can be any valid SDL/UPL identifier, one byte in length, a CHARACTER data type. After the FIND_DUPLICATE_CHARACTERS verb is performed, the value contained in <character-identifier> is the duplicate character found. For example, if the value equals the character A, the FIND_DUPLICATE_CHARACTER verb has found at least three consecutive characters equal to the character A.

reference-identifier-2
: This field can be any valid SDL/UPL reference identifier. After the FIND_DUPLICATE_CHARACTERS verb is performed, the value of <reference-identifier-2> is the character string of <reference-identifier-1. It ends immediately prior to the first duplicate character string.

Example:

```
Consider the character string:  "THIS IS THE PLAAAAACE"

FIND_DUPLICATE_CHARACTERS verb returns the following values:

   reference-identifier-1 = "CE"

   count-identifier = +0000005

   character-identifier = "A"

   reference-identifier-2 = "THIS IS THE PL"
```

Example Program:

```
DECLARE
        ACCEPT_FIELD      CHARACTER (69),
        REFERENCE_1       REFERENCE,
        REFERENCE_2       REFERENCE,
        COUNT             FIXED,
        CHARACTER_FIELD   CHARACTER (1);

DO FOREVER;

   DISPLAY ("ENTER A CHARACTER STRING OR ENTER BYE TO GO TO EOJ");
   ACCEPT ACCEPT_FIELD;
   IF ACCEPT_FIELD = "BYE" THEN STOP;
   REFER REFERENCE_1 TO ACCEPT_FIELD;

   FIND_DUPLICATE_CHARACTERS (REFERENCE_1,COUNT,CHARACTER_FIELD,
                              REFERENCE_2);
   DISPLAY ("THE RESULT OF REFERENCE_1 IS " CAT REFERENCE_1);
   DISPLAY ("THE DUPLICATE CHARACTER IS " CAT CHARACTER_FIELD);
   DISPLAY ("THE DUPLICATE CHARACTER APPEARS " CAT
            CONVERT (COUNT, CHARACTER) CAT
            " NUMBER OF TIMES");
   DISPLAY ("THE RESULT OF REFERENCE_2 IS " CAT REFERENCE_2);

END;

FINI;

% This example program accepts a character string from the
% ODT and locates any duplicate characters.  Using the
% FIND_DUPLICATE_CHARACTERS verb, the values of identifiers
% REFERENCE_1 and REFERENCE_2 are displayed.  Also, the
% duplicate character and number of times that the duplicate
% character appears is displayed.  Entering BYE terminates
% the program.
```

FINI

The FINI verb notifies the SDL/UPL compiler that this is the end of the source images to be compiled.

The FINI verb is optional. If the FINI verb is not specified, the SDL/UPL compiler uses the end-of-file record in the source file as the end of the source images.

SDL and UPL Syntax:

```
── FINI ──────────────────────────────────────────────────────────┤
```

Example:

```
DECLARE  A  CHARACTER (1);     % The FINI verb indicates the end of
A := "A";                      % source file to the SDL/UPL compiler.
DISPLAY (A);
STOP;
FINI;
```

FREEZE_PROGRAM

The FREEZE_PROGRAM verb prevents the program from being rolled out (moved to disk) during program execution. The MCP keeps the run structure of the program and saves space in the same memory location, regardless of the situation, until end of job or until the program performs the THAW_PROGRAM verb.

SDL and UPL Syntax:

```
—— FREEZE_PROGRAM; ——————————————————————————————————————————|
```

Example:

```
FREEZE_PROGRAM;
```

GROW

The GROW verb causes the array bound of the specified paged array to be dynamically increased by the value of <increase-amount>. The value of <increase-amount> cannot be negative and the resulting array bound cannot be larger than 16,777,215 (@(4)FFFFFF@) bytes.

Paged arrays grow by adding more pages to the array.

SDL and UPL Syntax:

```
—— GROW ( <paged-array-identifier>, <increase-amount> ) ; ——————————————————|
```

Syntax Semantics:

page-array-identifier
: This identifier can be any valid SDL/UPL paged array.

increase-amount
: This field can be any valid SDL/UPL literal, identifier or expression that returns a 24-bit binary value and specifies the number of elements to be added to the paged array.

Examples:

```
GROW (A, 10);           % Causes 10 elements to be added to
                        % the paged array A.

GROW (B, (BUMP X));     % Causes X + 1 elements to be added to
                        % the paged array B.
```

GROW

Example Program:

```
DECLARE PAGED (2)  CHAR_ARRAY (1)      CHARACTER (1),
                   INPUT_CHAR          CHARACTER (1),
                   COUNT               FIXED,
                   D_FIELD             CHARACTER (10);
D_FIELD := "";
COUNT := 0;
DO FOREVER;
   DISPLAY ("ENTER ONE CHARACTER OR ENTER BYE TO GO TO EOJ");
   ACCEPT INPUT_CHAR;
   IF INPUT_CHAR = "B" OR ((BUMP COUNT) > 9)
   THEN DO;
           DISPLAY ("GOOD BYE");
           STOP;
        END;

   GROW (CHAR_ARRAY, 1);  % Causes one element to be added to the
                          % paged array CHAR_ARRAY.

   CHAR_ARRAY (COUNT) := INPUT_CHAR;
   SUBSTR (D_FIELD,COUNT,1) := CHAR_ARRAY (COUNT);
   DISPLAY ("THE ARRAY EQUALS " CAT D_FIELD);
END;

FINI;

% This example program accepts a character from the ODT and
% causes the paged array to grow by one character to include the
% character.  The resulting paged array is displayed on the ODT.
% If more than 10 characters are entered, the program goes to
% end of job.
```

HALT

The HALT verb causes <halt-value> to be stored in the T-register and the M-machine halt instruction to be performed. The T-register can be examined on the console panel of the B 1000 computer system. The M-machine halt instruction stops the B 1000 processor.

SDL Syntax:

— HALT (<halt-value>) ; —

Syntax Semantics:

halt-value

This field can be any SDL literal, identifier, or expression and specifies the value to be loaded into the T-register. If <halt-value> is longer than 24 bits, only the leftmost 24 bits are stored. If <halt-value> is less than 24 bits, <halt-value> is stored in the T-register, right-justified with leading zeros.

Example:

```
DECLARE  X  BIT (24);    % Causes the value @00000A@ to be stored
X := 10;                 % into the T-register and the M-machine
HALT (X);                % halt instruction to be performed.
```

HASH__CODE

The HASH__CODE verb causes a 24-bit value to be returned. This value is computed from the length of the characters in <hash-code-value>. If the character string is longer than 15 characters, only the leftmost 15 characters are used.

To be effective, the value returned by the HASH__CODE verb must be used with a number that is divisible by a prime number. The prime number determines the logical hash-table size. Furthermore, <hash-code-value> modulo a prime number is the most effective hash-table index.

SDL and UPL Syntax:

```
—— HASH_CODE (<hash-code-value>) ——————————————————————————————————|
```

Syntax Semantics:

hash-code-value
: This field can be any valid SDL/UPL literal, identifier, or expression that returns a character value and specifies the value to be hashed.

Examples:

```
X := HASH_CODE ("JOHN DOE") MOD 13;      % Hashes the literal JOHN
                                         % DOE and assigns the
                                         % resulting value, modulo
                                         % 13, to the identifier X.

Y := HASH_CODE (CHARACTERS) MOD 29;      % Hashes the identifier
                                         % CHARACTERS and assigns the
                                         % resulting value, modulo
                                         % 29, to identifier Y.
```

Example Program:

```
DECLARE    CHARACTERS      CHARACTER (15),
           HASH_RESULT     BIT (24);

DO FOREVER;
   DISPLAY ("ENTER THE CHARACTERS TO BE HASHED OR ENTER BYE FOR EOJ");
   ACCEPT CHARACTERS;
   IF CHARACTERS = "BYE" THEN DO;
                                   DISPLAY ("GOOD BYE");
                                   STOP;
                                END;

   HASH_RESULT := HASH_CODE (CHARACTERS);

   DISPLAY ("THE HASH RESULT IS " CAT CONVERT(HASH_RESULT,CHARACTER));
END;

FINI;

% This example program accepts from the ODT up to 15 characters and
% uses the HASH_CODE verb on the accepted characters.  The result of
% hashing the characters is displayed on the ODT.
```

INITIALIZE__VECTOR

The INITIALIZE__VECTOR verb initializes the tables used by the SORT program.

This verb is for SORT program use only.

SDL Syntax:

```
—— INITIALIZE_VECTOR ( <table-address> ) ; ——————————————————————————————————|
```

Syntax Semantics:

table-address
: This field can be any SDL literal, identifier, or expression that returns a 24-bit value and specifies the address of the table containing the vector addresses, the vector level-1 address, the key table address, and the vector limit address.

LAST_LIO_STATUS

The LAST_LIO_STATUS verb returns a bit value with a length equal to the RS_LAST_LIO_STATUS_SIZE field in the run structure nucleus of the SDL program. This value represents the current status of logical input/output (I/O) operation for the SDL program.

SDL Syntax:

```
—— LAST_LIO_STATUS ——————————————————————————————————————————|
```

Example:

```
DECLARE LAST_IO_STATUS  BIT (24);
LAST_IO_STATUS := LAST_LIO_STATUS;
```

Example Program:

```
FILE PORTFILE (DEVICE = PORT,
               RECORDS = 80/1,
               HOST_NAME = "B1000");

RECORD 01 STATUS_MASK_EXCEPTION     BIT (24),
          02 ANY_EXCEPTION          BIT (1),
          02 FILLER                 BIT (4),
          02 INVALID_SUBPORT_INDEX  BIT (1),
          02 FILLER                 BIT (1),
          02 IO_ERROR               BIT (1),
          02 FILLER                 BIT (1),
          02 LOGICAL_EOF            BIT (1),
          02 FILLER                 BIT (1),
          02 SUBPORT_STATE_CHANGE   BIT (1),
          02 FILLER                 BIT (3);

DECLARE BUFFER   CHARACTER (80),
        X        STATUS_MASK_EXCEPTION,
        MASK     BIT (24); % THIS IS THE RESULT MASK

OPEN PORTFILE WITH INPUT, OUTPUT;

MASK := @FFFFFF@; % REPORTS ALL EXCEPTIONS
DO FOREVER;
   READ  PORTFILE (BUFFER) WITH RESULT_MASK MASK;
         ON EXCEPTION DO;
```

LAST_LIO_STATUS

```
                        DISPLAY "EXCEPTION ON READ OF PORTFILE";
                        X := LAST_LIO_STATUS; % IDENTIFIER X CONTAINS
                                              % ALL EXCEPTIONS WHICH
                                              % OCCURRED.
                        IF SUBBIT (X, 6, 1) = 1
                           THEN DISPLAY "INVALID SUBPORT INDEX";
                     END;
   WRITE PORTFILE (BUFFER);
         ON EXCEPTION DISPLAY "EXCEPTION ON WRITE OF PORTFILE";
   DISPLAY (BUFFER);
END;
FINI;

% This example program uses the LAST_LIO_STATUS verb to
% assign all the exceptions for a read operation to a BNA
% port file.  The program reads from the port file, writes
% the same message back (echo) to the port file, and displays
% the message read/written on the ODT.
```

LENGTH

The LENGTH verb returns a 24-bit value, which contains the number of units in <identifier>, where unit is either of the following:

1. The number of characters if <identifier> has a data type of CHARACTER.
2. The numbers of bits if <identifier> has a data type of FIXED or BIT.

SDL and UPL Syntax:

```
── LENGTH (<identifier>) ──────────────────────────────────────────┤
```

Syntax Semantics:

identifier
    This field can be any valid SDL/UPL identifier or expression that returns an addressable value.

Examples:

```
X := LENGTH ("23");     % The identifier X is assigned a 24-bit
                        % value equal to 2 or @(4)000002@.

X := LENGTH (Y);        % The identifier X is assigned a 24-bit
                        % value equal to the length of Y.
```

LENGTH

Example Program:

```
DECLARE      CHARACTERS                 CHARACTER (1950),
             LENGTH_OF_CHARACTERS       BIT (24),
             COUNTER                    FIXED;

DO FOREVER;
   DISPLAY ("ENTER ANY NUMBER OF CHARACTERS OR ENTER BYE FOR EOJ");
   ACCEPT CHARACTERS;
   COUNTER := 0;
   DO CHARACTER_LOOP FOREVER;
     IF SUBSTR (CHARACTERS, COUNTER, 1) = " "  OR COUNTER > 1948
     THEN IF SUBSTR (CHARACTERS, 0, COUNTER) = "BYE"
          THEN DO;
                  DISPLAY ("GOOD BYE");
                  STOP;
               END;
          ELSE DO;
                  LENGTH_OF_CHARACTERS :=
                       LENGTH (SUBSTR (CHARACTERS, 0, COUNTER));
                  DISPLAY ("THE LENGTH OF THE CHARACTERS ENTERED IS "
                            CAT DECIMAL (LENGTH_OF_CHARACTERS, 8));
                  UNDO CHARACTER_LOOP;
               END;
     BUMP COUNTER;
   END CHARACTER_LOOP;
END;

FINI;

% This example program accepts a character field from the ODT
% and uses the LENGTH verb to calculate the number of characters
% entered.  If "BYE" is entered the program goes to end of job.
```

LIMIT_REGISTER

The LIMIT_REGISTER verb returns a 24-bit value which is the base-relative address of the Run Structure Nucleus for the program.

SDL and UPL Syntax:

```
—— LIMIT_REGISTER ——————————————————————————————————————|
```

Example:

```
DECLARE X  BIT (24);
X := LIMIT_REGISTER;       % Assigns to identifier X a 24-bit value
                           % which represents the limit register of
                           % the run structure nucleus in the program.
```

Example Program:

```
DISPLAY ("THE ADDRESS OF THE RUN STRUCTURE NUCLEUS IN THIS PROGRAM IS "
        CAT DECIMAL (LIMIT_REGISTER, 8));
STOP;
FINI;

% This example program displays on the ODT the base-relative address
% of the program's run structure nucleus and goes to end of job.
```

LOCATION

The LOCATION verb returns a bit value that is the base-relative address of the specified identifier, array-identifier, or procedure-identifier.

When a procedure-identifier is specified, a 36-bit value is returned. This 36-bit value contains, as the first four bits, the address type which is equal to @F@ or @(1)1111@. This value designates that this 36-bit value applies to a procedure identifier. Also, included in this 36-bit value is the page, segment, and displacement of the specified procedure.

The following is the format of the 36-bit value for a procedure identifier.

```
01 PROCEDURE_ADDRESS     BIT (36),

   03 ADDRESS_TYPE       BIT (4),  % Contains the value @F@
   03 SEGMENT_NUMBER     BIT (6),
   03 PAGE_NUMBER        BIT (6),
   03 DISPLACEMENT       BIT (20);
```

When an identifier or array-identifier is specified, a 16-bit value is returned. The first two bits of this field is the address type and equals @(1)00@ or @(2)0@. This 2-bit value designates that the remaining 16-bit value represents an identifier or an array. The remaining information includes the lexic level and the occurrence number within the lexic level for the identifier or array.

The following is the format of the 16-bit value.

```
01 IDENTIFIER_OR_ARRAY_ADDRESS  BIT (16),

   03 ADDRESS_TYPE              BIT (2),  % Contains the value @(2)0@
   03 LEXIC_LEVEL               BIT (4),
   03 OCCURRENCE_NUMBER         BIT (10);
```

SDL and UPL Syntax:

```
── LOCATION ( ─────┬── <identifier> ──────────────┬── ) ──────────┤
                   ├── <array-identifier> ────────┤
                   └── <procedure-identifier> ────┘
```

Syntax Semantics:

identifier
This identifier can be any valid SDL/UPL identifier.

array-identifier
This array identifier can be any valid SDL/UPL array identifier.

procedure-identifier
This procedure identifier can be any valid SDL/UPL procedure identifier. This procedure must be declared as a FORWARD procedure if a recompilation or create-master compilation is to be performed.

LOCATION

Examples:

```
DECLARE    X              BIT (16),
           Y              BIT (36),
           IDENTIFIER     CHARACTER (10),
           ARRAY (20)     BIT (24);

X := LOCATION (IDENTIFIER);        % Assigns to identifier X a 16-bit
                                   % value with @(1)00@ as the first
                                   % two bits, followed by a 4-bit
                                   % lexic-level number equal to
                                   % @(1)0000@ and a 10-bit occurrence
                                   % number equal to @(1)0000000010@.

X := LOCATION (ARRAY);             % Assigns to identifier X a 16-bit
                                   % value with @(1)00 as the first
                                   % two bits, followed by a 4-bit
                                   % lexic-level number equal to
                                   % @(1)0000@ and a 10-bit occurrence
                                   % number equal to @(1)0000000011@.

Y := LOCATION (PROCEDURE_ONE);     % Assigns to identifier Y a 36-bit
                                   % value with @F@ as the first four
                                   % bits, followed by a 6-bit segment
                                   % number, a 6-bit page number and a
                                   % 20-bit displacement number of
                                   % procedure PROCEDURE_ONE.
```

Example Program:

```
SEGMENT (ZERO);

PROCEDURE DISPLAY_ARRAY_AND_FIELD;

   DECLARE   01 LOC_OF_ARRAY_OR_FIELD     BIT (16),
                03 ADDRESS_TYPE_AF        BIT (2),
                03 LEXIC_LEVEL            BIT (4),
                03 OCCURRENCE_NUMBER      BIT (10),
             ARRAY (10)                   CHARACTER (10),
             FIELD                        FIXED;

    LOC_OF_ARRAY_OR_FIELD := LOCATION (ARRAY);

    DISPLAY ("THE ADDRESS TYPE OF THE ARRAY IS " CAT
             CONVERT (ADDRESS_TYPE_AF, CHARACTER));
    DISPLAY ("THE LEXIC LEVEL OF THE ARRAY IS " CAT
             CONVERT (LEXIC_LEVEL, CHARACTER));
    DISPLAY ("THE OCCURRENCE NUMBER OF THE ARRAY IS " CAT
             CONVERT (OCCURRENCE_NUMBER, CHARACTER));

    LOC_OF_ARRAY_OR_FIELD := LOCATION (FIELD);
```

LOCATION

```
     DISPLAY ("THE ADDRESS TYPE OF FIELD IS " CAT
              CONVERT (ADDRESS_TYPE_AF, CHARACTER));
     DISPLAY ("THE LEXIC LEVEL OF FIELD IS " CAT
              CONVERT (LEXIC_LEVEL, CHARACTER));
     DISPLAY ("THE OCCURRENCE NUMBER OF FIELD IS " CAT
              CONVERT (OCCURRENCE_NUMBER, CHARACTER));

END DISPLAY_ARRAY_AND_FIELD;
%
SEGMENT (ONE);
%
PROCEDURE DISPLAY_PROCEDURE;

   DECLARE  01 LOC_OF_PROCEDURE         BIT (36),
               03 ADDRESS_TYPE_P        BIT (4),
               03 SEGMENT_NUMBER        BIT (6),
               03 PAGE_NUMBER           BIT (6),
               03 DISPLACEMENT_NUMBER   BIT (20);

   LOC_OF_PROCEDURE := LOCATION (DISPLAY_PROCEDURE);

   DISPLAY ("THE ADDRESS TYPE OF DISPLAY_PROCEDURE IS " CAT
            CONVERT (ADDRESS_TYPE_P, CHARACTER));
   DISPLAY ("THE SEGMENT NUMBER OF DISPLAY_PROCEDURE IS " CAT
            CONVERT (SEGMENT_NUMBER, CHARACTER));
   DISPLAY ("THE PAGE NUMBER OF DISPLAY_PROCEDURE IS " CAT
            CONVERT (PAGE_NUMBER, CHARACTER));
   DISPLAY ("THE DISPLACEMENT OF DISPLAY_PROCEDURE IS " CAT
            CONVERT (DISPLACEMENT_NUMBER, CHARACTER));

END DISPLAY_PROCEDURE;
%
% MAIN PROGRAM BEGINS HERE
%
SEGMENT (TWO);

DISPLAY_ARRAY_AND_FIELD;

DISPLAY_PROCEDURE;

STOP;

SEGMENT (ZERO);

FINI;

% This example program displays the location of ARRAY, FIELD,
% and DISPLAY_PROCEDURE and goes to end of job.
```

Output from Example Program:

```
LOCATIONO =7523 BOJ. PP=4, MP=4 TIME = 15:03:10.7
% LOCATIONO =7523 THE ADDRESS TYPE OF THE ARRAY IS 0
% LOCATIONO =7523 THE LEXIC LEVEL OF THE ARRAY IS 1
% LOCATIONO =7523 THE OCCURRENCE NUMBER OF THE ARRAY IS 006

% LOCATIONO =7523 THE ADDRESS TYPE OF FIELD IS 0
% LOCATIONO =7523 THE LEXIC LEVEL OF FIELD IS 1
% LOCATIONO =7523 THE OCCURRENCE NUMBER OF FIELD IS 007

% LOCATIONO =7523 THE ADDRESS TYPE OF DISPLAY_PROCEDURE IS F
% LOCATIONO =7523 THE SEGMENT NUMBER OF DISPLAY_PROCEDURE IS 02
% LOCATIONO =7523 THE PAGE NUMBER OF DISPLAY_PROCEDURE IS 00
% LOCATIONO =7523 THE DISPLACEMENT OF DISPLAY_PROCEDURE IS 00000
LOCATIONO =7523 EOJ. TIME = 15:03:33.1
```

MAKE_DESCRIPTOR

The MAKE_DESCRIPTOR verb replaces the current entry on the evaluation stack with <descriptor>. If the name-value bit of <descriptor> on the evaluation stack is set, the value of <descriptor> is removed from the value stack.

The DESCRIPTOR verb can appear as the object of a replacement, as long as the descriptor created generates an address.

SDL Syntax:

```
—— MAKE_DESCRIPTOR (<descriptor>) ——————————————————————————————|
```

Syntax Semantics:

descriptor
    This field can be any valid SDL expression that returns a descriptor.

Examples:

```
MAKE_DESCRIPTOR (DESCRIPTOR (X)) = X,
   where X is non-self-relative.

MAKE_DESCRIPTOR (VALUE_DESCRIPTOR (E)) = E,
   where E generates an address.

VALUE_DESCRIPTOR (MAKE_DESCRIPTOR (E)) := E.
   where the value of E is a valid address generator.
```

MAKE_READ_ONLY

The MAKE_READ_ONLY verb applies only to paged arrays and marks the specified page number of a paged array as READ_ONLY. All pages within a paged array are marked as READ_WRITE by default. Once a page is marked as READ_ONLY, that page is not copied to disk each time it is overlaid by the MCP. The programmer is responsible for insuring that information written to a page, within a paged array, be performed when the page is not marked READ_ONLY. Refer to the MAKE_READ_WRITE verb to mark a paged array as READ_WRITE.

The programmer must calculate <page-number>, and also must ensure that <page-number> is a valid page number. No syntax checking is performed on the value used to reference a page number within a paged array.

SDL and UPL Syntax:

```
── MAKE_READ_ONLY (<paged-array-identifier>), <page-number>); ──────────────┤
```

Syntax Semantics:

paged-array-identifier
: This field can be any valid SDL/UPL paged-array identifier and specifies the paged array to be marked as READ_ONLY.

page-number
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a 24-bit binary value and specifies the page number within a paged array.

Examples:

```
DECLARE PAGED (32) P (1024) BIT (30),
                   I        FIXED;

MAKE_READ_ONLY (P, 1);         % Makes page number one of the paged
                               % array P a READ_ONLY page.

MAKE_READ_ONLY (P, I);         % Makes the page number specified by the
                               % value of I a READ_ONLY page.

MAKE_READ_ONLY (P, BUMP I);    % Makes the page number specified by
                               % the value of I +1 a READ_ONLY page.
```

Example Program:

```
DECLARE  PAGED (2) P (32)    FIXED,
         I                   FIXED,
         ODT_INPUT           CHARACTER (5);

DO FOREVER;
   MAKE_READ_ONLY (P, BUMP I);
   IF I = 15 THEN UNDO;
END;
```

MAKE_READ_ONLY

```
DO FOREVER;
   DISPLAY ("ENTER READ, ENTER WRITE, OR ENTER BYE TO GO TO EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN STOP;
   IF ODT_INPUT = "READ"
   THEN DO;
           DISPLAY ("ENTER AN ELEMENT NUMBER BETWEEN 0 AND 31");
           ACCEPT ODT_INPUT;
           I := CONVERT (ODT_INPUT, FIXED);
           IF I > 31
           THEN DISPLAY ("NUMBER ENTERED IS TOO LARGE");
           ELSE DISPLAY (DECIMAL (P (I), 8));
        END;
   ELSE IF ODT_INPUT = "WRITE"
        THEN DO;
                DISPLAY ("ENTER AN ELEMENT NUMBER BETWEEN 0 AND 31");
                ACCEPT ODT_INPUT;
                I := CONVERT (ODT_INPUT, FIXED);
                IF I > 31
                THEN DISPLAY ("NUMBER ENTERED IS TOO LARGE");
                ELSE DO;
                        DISPLAY ("ENTER A NUMBER");
                        MAKE_READ_WRITE (P, I/2);
                        ACCEPT ODT_INPUT;
                        P (I) := CONVERT (ODT_INPUT,FIXED);
                        MAKE_READ_ONLY (P, I/2);
                     END;
             END;
        ELSE DISPLAY ("INCORRECT COMMAND -- TRY READ, WRITE, OR BYE");
END;

STOP;

FINI;

% This example program illustrates the use of the MAKE_READ_ONLY
% and MAKE_READ_WRITE verbs on paged arrays.  The program first
% accepts from the ODT the entries "READ", "WRITE", or "BYE".  If
% "BYE" is entered the program goes to end of job.  If "READ" is
% entered, the program then accepts from the ODT an element number
% between 0 and 31 and displays the contents of that element in
% the array.  If "WRITE" is entered, the program accepts from the
% ODT the element number between 0 and 31 and then a 5-character
% value to be placed into that element within the paged array.
%
% The MAKE_READ_ONLY verb is used to initially make all the pages
% in the paged array READ_ONLY and, also, after an element in the
% paged array has been changed.  The MAKE_READ_WRITE verb is used
% to make an element in the paged array READ_WRITE in order to
% change the value of the element.
```

MAKE_READ_WRITE

The MAKE_READ_WRITE verb changes the status of the page within a paged array specified by <page-number> to READ_WRITE. If the status of a page is READ_WRITE, the page is copied to disk each time it is overlaid by the MCP.

The user must calculate <page-number>, and also must ensure that <page-number> is valid. No syntax checking is performed by the SDL/UPL compiler to verify that <page-number> is valid.

Unless a page has been marked as READ_ONLY by the MAKE_READ_ONLY verb, a status of READ_WRITE is the default for all pages within a paged array. The MAKE_READ_WRITE verb is only needed to override READ_ONLY status set by the MAKE_READ_ONLY verb.

SDL and UPL Syntax:

```
—— MAKE_READ_WRITE (<paged-array-identifier>, <page-number>); ——————————————|
```

Syntax Semantics:

paged-array-identifier
: This field can be any valid SDL/UPL paged-array identifier and specifies the paged array to be marked as READ_WRITE.

page-number
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a 24-bit binary number and specifies the page within a paged array.

Examples:

```
DECLARE PAGED (32) P (1024) BIT (30),
                   I        FIXED;

MAKE_READ_WRITE (P, 1);       % Makes page number 1 of the paged
                              % array P a READ_ONLY page.

MAKE_READ_WRITE (P, I);       % Makes the page number specified by
                              % the value of I a READ_ONLY page.

MAKE_READ_WRITE (P, BUMP I);  % Makes the page number specified by
                              % the value of I + 1 a READ_ONLY page.
```

Example Program:

Refer to the Example Program for the MAKE_READ_ONLY verb.

MESSAGE_COUNT

The MESSAGE_COUNT verb scans the specified queue file and determines the number of messages currently in the queue. This number is stored in <identifier> with a FIXED data type.

When the queue file specified is a queue file family, the MESSAGE_COUNT verb returns an array of FIXED values, one for each file in the family. The programmer must ensure that <identifier> is large enough to hold the generated value.

SDL and UPL Syntax:

```
—— MESSAGE_COUNT ( <queue-file-id>, <identifier> ); ——————————————|
```

Syntax Semantics:

queue-file-id
: This field can be any valid SDL/UPL file identifier declared with a device type equal to QUEUE and specifies the queue file name to obtain the message count.

identifier
: This field can be any valid SDL/UPL identifier and specifies the destination field for the number of messages.

Examples:

```
DECLARE     X (5) FIXED,
            Y     FIXED;

FILE QUEUE_FILE (DEVICE=QUEUE),
     QUEUE_FAMILY_5 (DEVICE = QUEUE (5));

MESSAGE_COUNT (QUEUE_FILE, Y);          % Stores the number of messages
                                        % queued for QUEUE_FILE into
                                        % identifier Y.

MESSAGE_COUNT (QUEUE_FAMILY_5, X);      % Stores the number of messages
                                        % queued for each file within
                                        % the QUEUE_FAMILY_5 into array
                                        % X.
```

MESSAGE_COUNT

Example Program:

```
DECLARE    NUMBER_OF_MESSAGES  FIXED,
           COUNTER             FIXED;

FILE  QUEUE (DEVICE = QUEUE (10),
             OPEN_OPTION = OUTPUT,
             RECORDS = 10,
             BUFFERS = 2);

COUNTER := 0;
DO FOREVER;
   WRITE QUEUE (COUNTER);

   MESSAGE_COUNT (QUEUE, NUMBER_OF_MESSAGES);

   DISPLAY ("THE NUMBER OF MESSAGES QUEUED EQUALS " CAT
            CONVERT (NUMBER_OF_MESSAGES, CHARACTER));
   IF ((BUMP COUNTER) > 9) THEN DO;
                                    DISPLAY ("GOOD BYE");
                                    STOP;
                                 END;
END;

FINI;

% This example program writes a message to the file labeled QUEUE
% and uses the MESSAGE_COUNT verb to interrogate the number of
% messages in the queue file.  The number of messages is displayed
% on the ODT.
```

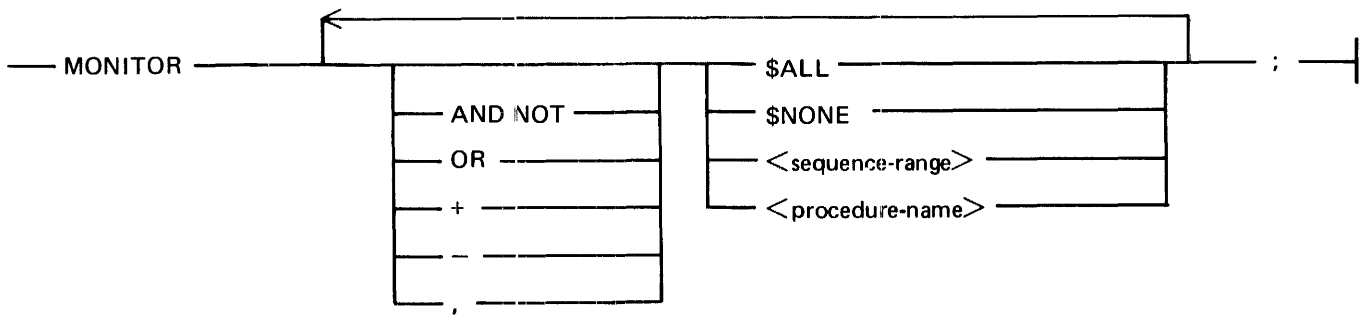
Output from Example Program:

```
MESSAGEO =7076 BOJ. PP=4, MP=4 TIME = 08:21:15.2
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000001
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000002
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000003
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000004
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000005
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000006
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000007
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000008
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000009
% MESSAGEO =7076 THE NUMBER OF MESSAGES QUEUED EQUALS +0000010
% MESSAGEO =7076 GOOD BYE
MESSAGEO =7076 EOJ. TIME = 08:21:36.5
```

MONITOR

The MONITOR verb specifies which procedures are candidates to be monitored.

SDL Syntax:



Syntax Semantics:

AND NOT
: The keywords AND NOT cause the sequence numbers specified by <sequence-range> or the procedures specified by <procedure-name> not to be monitored.

OR
: The keyword OR causes the sequence numbers specified by <sequence-range> or the procedures specified by <procedure-name> to be monitored.

+
: The key symbol + causes the sequence numbers specified by <sequence-range> or the procedures specified by <procedure-name> to be monitored.

–
: The key symbol – causes the sequence numbers specified by <sequence-range> or the procedures specified by <procedure-name> not to be monitored.

,
: The keysymbol , causes the sequence numbers specified by <sequence-range> or the procedures specified by <procedure-name> to be monitored.

$ALL
: The keyword $ALL causes all of the procedures to be monitored.

$NONE
: The keyword $NONE causes no procedures to be monitored.

sequence-range
: This field can be any sequence range of sequence numbers within the SDL/UPL source file. It specifies the sequence range for monitoring a designated procedure. The following is the format for <sequence-range>, where bbbbbbbb specifies the beginning sequence number and eeeeeeee specifies the ending sequence number.

    bbbbbbbb-eeeeeeee

procedure-name
: This field can be any procedure identifier within the SDL/UPL program that is marked to be monitored and specifies that this procedure is to be monitored.

MONITOR

Example 1:

```
MONITOR ("$ALL");                     % Causes all procedures that are
                                      % candidates for monitoring to be
                                      % monitored.
```

Example 2:

```
MONITOR ("$NONE");                    % Causes no procedures to be
                                      % monitored.
```

Example 3:

```
MONITOR ("X1, X2");                   % Causes procedures X1 and X2 to be
                                      % monitored.
```

Example 4:

```
MONITOR ("00000000-01999999"); % Causes all procedures between
                               % sequence numbers 00000000 and
                               % 01999999 to be monitored.
```

Example 5:

```
MONITOR ("X1 AND NOT X2");            % Causes procedure X1 to be monitored
                                      % but not procedure X2.
```

Example Program:

```
 DECLARE  ODT_INPUT  CHARACTER (3);
$ MONITOR
 PROCEDURE COUNT;
   DECLARE COUNT FIXED;
   DISPLAY (CONVERT ((BUMP COUNT), CHARACTER));
 END COUNT;

 DO FOREVER;
  DISPLAY ("ENTER YES TO MONITOR PROCEDURE AGAIN OR ENTER BYE FOR EOJ");
  ACCEPT ODT_INPUT;
  IF ODT_INPUT = "BYE" THEN DO;
                               DISPLAY ("GOOD BYE");
                               STOP;
                             END;
  IF ODT_INPUT = "YES" THEN MONITOR_SET ("COUNT");
                       ELSE MONITOR_RESET ("COUNT");

  COUNT;
 END;
 FINI;
```

M_MEM_SIZE

The M_MEM_SIZE verb returns a 24-bit value which is the M-memory size in bits, of the B 1720 computer system.

The M_MEM_SIZE verb is only valid for the B 1720 series computer.

SDL and UPL Syntax:

```
—— M_MEM_SIZE ——————————————————————————————————————————————————————|
```

Example:

```
DECLARE  MEMORY  BIT (24);    % Identifier MEMORY is assigned the
MEMORY := M_MEM_SIZE;         % value of the memory size of the B1720
                              % computer system.
```

Example Program:

```
DISPLAY ("THE M-MEMORY SIZE EQUALS @" CAT
         CONVERT ((M_MEM_SIZE / 8), CHARACTER) CAT "@ BYTES");
STOP;
FINI;
```

Output from Example Program:

```
M_MEM_SIZO =6234 BOJ. PP=4, MP=4 TIME = 10:37:11.4
% M_MEM_SIZO =6234 THE M-MEMORY SIZE EQUALS @0060000@ BYTES
M_MEM_SIZO =6234 EOJ. TIME = 10:37:16.7
```

NAME__OF__DAY

The NAME__OF__DAY verb returns a left-justified, 9-character string which is the name of the current system day of the week. The seven possible values are MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, and SUNDAY.

SDL and UPL Syntax:

```
—— NAME_OF_DAY ——————————————————————————————————————|
```

Example:

```
DECLARE  NAME  CHARACTER (9);
NAME := NAME_OF_DAY;

% If the current system day name is WEDNESDAY, then
% NAME has the following bit and hexadecimal values.
%
% NAME = @(4)E6C5C4D5C4E2C4C1E8@
%      = "WEDNESDAY"
```

Example Program:

```
DISPLAY ("TODAYS DAY NAME IS " CAT NAME_OF_DAY);
STOP;
FINI;
```

Output from Example Program:

```
NAMEOFDAYO =5598 BOJ. PP=4, MP=4 TIME = 08:00:45.9
% NAMEOFDAYO =5598 TODAYS DAY NAME IS FRIDAY
NAMEOFDAYO =5598 EOJ. TIME = 08:00:50.5
```

NAME_STACK_TOP

The NAME_STACK_TOP verb returns a 24-bit, self-relative value with a BIT data type. This 24-bit value is the base-relative address of the top of the name stack.

SDL Syntax:

```
—— NAME_STACK_TOP ——————————————————————————————————————————————|
```

Example:

```
DECLARE  NAME_STACK_ADDR  BIT (24);  % Identifier NAME_STACK_ADDR
NAME_STACK_ADDR := NAME_STACK_TOP;   % is assigned the address of
                                     % the top of the name stack.
```

Example Program:

```
DISPLAY ("THE ADDRESS OF THE TOP OF THE NAME STACK EQUALS " CAT
        CONVERT (NAME_STACK_TOP, CHARACTER));
STOP;
FINI;
```

Output from Example Program:

```
NAMESTACKO =5601 BOJ. PP=4, MP=4 TIME = 08:05:47.8
% NAMESTACKO =5601 THE ADDRESS OF THE TOP OF THE NAME STACK
     EQUALS 0027D0
NAMESTACKO =5601 EOJ. TIME = 08:05:51.8
```

NEXT_ITEM

The NEXT_ITEM verb causes the length field of the descriptor, represented by <identifier>, to be added to the address field of that descriptor. This modified descriptor is put back onto the name stack and is also moved to the top of the evaluation stack. This modified descriptor is the load address of the new item described by <identifier>.

SDL Syntax:

── NEXT_ITEM (<identifier>) ──────────────────┤

Syntax Semantics:

identifier

This field can be any valid SDL simple identifier and specifies the name of the starting identifier.

Example:

```
DECLARE 01 CHAR_STRING  CHARACTER (1000),   % Causes the character D
           03 NEXT_CHAR CHARACTER (1);      % to be moved into the
NEXT_ITEM (NEXT_CHAR) := "D";               % second character of
                                            % CHAR_STRING.
```

NEXT_TOKEN

The NEXT_TOKEN verb returns the descriptor of the next token. This token can be an identifier, a number, or a special character. The descriptor of <result-reference-identifier> is also replaced by this descriptor. <first-character-address> is changed to point to the character which immediately follows this token. The NEXT_TOKEN verb expects that the <first-character-address> references a nonblank character.

SDL Syntax:

```
—— NEXT_TOKEN (<first-character-address>, <separator>, ——————┬— SET ———┬——>
                                                             └— RESET ——┘

>——— , <result-reference-identifier>) ————————————————————————————————————|
```

Syntax Semantics:

first-character-address
: This field can be any valid SDL identifier and specifies the address of the first character in the character string to be scanned.

separator
: This field can be a character string or a bit string with a length equal to eight bits and specifies the token separator. The SDL compiler uses the underscore (_) character. If no token separator is required, specify the character A.

SET
: The keyword SET allows the symbols 0 through 9 to be valid symbols. For example, the symbols 235AB are allowed.

RESET
: The keyword RESET does not allow the symbols 0 through 9 to be valid symbols. For example, the symbol 456DF is not allowed.

result-reference-identifier
: This identifier can be any valid SDL reference identifier. It specifies the name of the field in which to store the string of characters. It begins with <first-character-address> and ends with, but does not include, any <separator> encountered during the scan.

Example:

```
DECLARE FIRST_CHAR      REFERENCE,       % The identifier NEXT_CHAR
        RESULT          REFERENCE,       % is assigned the value
        CHAR_STRING     CHARACTER (15),  % "7".
        NEXT_CHAR       CHARACTER (15);
CHAR_STRING := "12345_789;ABCDE";
REFER FIRST_CHAR
TO SUBSTR (CHAR_STRING, 0, 1);
NEXT_CHAR :=
 DELIMITED_TOKEN (FIRST_CHAR, "_", SET, RESULT);
```

NEXT_TOKEN

Example Program:

```
DECLARE  ODT_INPUT         CHARACTER (50),
         RESULT            REFERENCE,
         FIRST_CHARACTER   REFERENCE;

DO FOREVER;
   DISPLAY ("ENTER ANY 50-CHARACTERS TO BE SCANNED OR ENTER BYE FOR"
            CAT " EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                             DISPLAY ("GOOD BYE");
                             STOP;
                          END;
   REFER FIRST_CHARACTER TO SUBSTR (ODT_INPUT, 0, 1);
   DISPLAY ("THE NEXT TOKEN EQUALS");

   DISPLAY (NEXT_TOKEN (FIRST_CHARACTER, "_", SET, RESULT));
   DISPLAY (FIRST_CHARACTER);
   DISPLAY (RESULT);
END;
FINI;

% This example program finds the first token of a 50-character
% message entered from the ODT and displays the token back on
% the ODT.  If BYE is entered, the program goes to end of job.
```

OPEN

The OPEN verb allows a program to explicitly open a data file.

The OPEN verb requests permission from the MCP to access a file and to make available the requested memory space. An implicit open is performed by the MCP when a program reads from or writes to a data file that has not been explicitly opened with the OPEN verb.

Buffer storage is allocated and file attributes are established when a file is opened. Memory storage utilization can be significantly optimized by delaying a file open operation until the file is needed.

The open attributes specified with the OPEN verb override any FILE declaration attributes. Attributes not specified in the OPEN verb maintain the status set in the FILE declaration, or the default status if not specified.

The NEW open attribute is only valid with the OUTPUT open attribute. If the OUTPUT open attribute is not specified when the NEW open attribute is specified, OUTPUT is assumed by default. Specifying the open attributes INPUT and NEW without specifying the OUTPUT open attribute generates a syntax error.

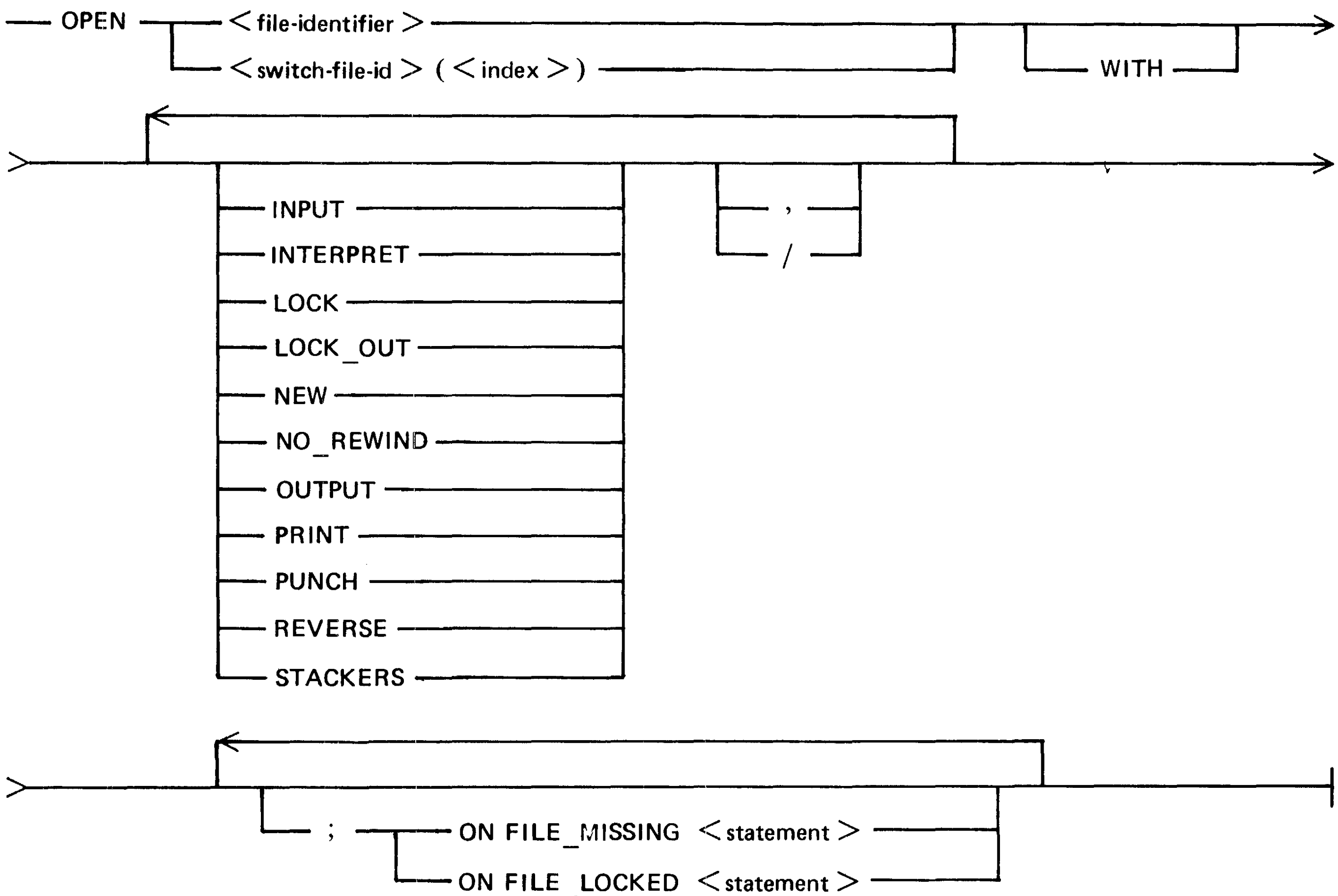
Specifying INPUT OUTPUT NEW is only valid with files whose access attribute is equal to RANDOM.

The LOCK open attribute protects the file from write operations by another program.

The LOCK_OUT open attribute protects the file from read operations as well as write operations by another program.

OPEN

SDL and UPL Syntax:



Syntax Semantics:

file-identifier
: This field can be any valid SDL/UPL file identifier and specifies the file to be opened.

switch-file-id
: This field can be any valid SDL/UPL switch-file identifier and specifies the switch file to be opened.

index
: This field can be any valid SDL/UPL identifier and specifies the number of the switch file to be opened.

INPUT
: The keyword INPUT causes the SDL/UPL program to open an existing file and allows the program to read from the file.

INTERPRET
: The keyword INTERPRET causes the SDL/UPL program to interpret card-image records as each is written. INTERPRET only affects files with a device type equal to DATA_RECORDER_80, PUNCH_PRINTER, READER_PUNCH, or READER_PUNCH_PRINTER.

OPEN

LOCK
The keyword LOCK prevents another program from opening the specified file with the OUTPUT open attribute. Opening the file with the INPUT open attribute by another program is allowed. Once the file is closed, the file can be opened by another program with the OUTPUT open attribute.

LOCK_OUT
The keyword LOCK_OUT prevents another program from opening the specified file with the INPUT or OUTPUT open attributes. Once the file is closed, the file can be opened by another program with the INPUT or OUTPUT open attributes.

NEW
The keyword NEW specifies that the file is to be created.

NO_REWIND
The keyword NO_REWIND applies to files with a device type equal to TAPE, TAPE_9, TAPE_7, TAPE_PE, and TAPE_NRZ and prevents the MCP from rewinding the tape file when an end-of-tape mark is encountered.

OUTPUT
The keyword OUTPUT allows the SDL/UPL program to write to an existing file.

PRINT
The keyword PRINT applies to files with a device type equal to DATA_RECORDER_80, PUNCH_PRINTER, READER_PUNCH, or READER_PUNCH_PRINTER and allows the SDL/UPL program to interpret and punch card-image records.

REVERSE
The keyword REVERSE applies to files with a device type equal to TAPE, TAPE_9, TAPE_7, TAPE_PE, and TAPE_NRZ and notifies the MCP that the tape file is to be written or read in reverse. The programmer must ensure that the tape file is positioned so that the backspacing operation can be performed. Read operations on a tape file, with the REVERSE open attribute specified, report the end-of-file (EOF) record when the beginning-of-tape (BOT) mark is encountered.

STACKERS
The keyword STACKERS applies to files with a device type equal to DATA_RECORDER_80, PUNCH_PRINTER, READER_PUNCH, or READER_PUNCH_PRINTER and allows the SDL/UPL program to specify that the stackers on the card device are to be used.

ON FILE_MISSING
The keywords ON FILE_MISSING cause the SDL/UPL program to perform the associated statement if the file specified is not present at the time the OPEN verb is performed.

ON FILE_LOCKED
The key words ON FILE_LOCKED cause the SDL/UPL program to perform the associated statement if the file specified is currently locked by another program. This can occur in either of the two following conditions:

1. The INPUT or OUTPUT open attributes were specified and another program has opened the same file with the LOCK-OUT open attribute.
2. The OUTPUT open attribute was specified and another program has opened the same file with the LOCK open attribute.

OPEN

statement
    This statement can be any valid SDL/UPL statement.

Examples:

```
OPEN CARD_FILE INPUT;

OPEN DISK_FILE INPUT OUTPUT NEW;
     ON FILE_MISSING DISPLAY ("FILE NOT PRESENT");

OPEN DISK_FILE INPUT LOCK;
     ON FILE_LOCKED DISPLAY ("FILE LOCKED");

OPEN TAPE_FILE NO_REWIND INPUT;

OPEN TAPE_FILE REVERSE OUTPUT;

OPEN CARD_FILE WITH STACKERS INPUT;

OPEN CARD_FILE WITH OUTPUT PUNCH INTERPRET;

OPEN DISK_FILE OUTPUT NEW;
     ON FILE_MISSING DISPLAY ("FILE NOT PRESENT");
     ON FILE_LOCKED DISPLAY ("FILE LOCKED");
```

OPEN

Example Program:

```
FILE DISKFILE (DEVICE = DISK,
               RECORDS =180/10);

ZIP "SO OPEN;";     % Sets the MCP OPEN option

OPEN DISKFILE WITH INPUT;
     ON FILE_MISSING
        DO;
           DISPLAY ("FILE DISKFILE NOT PRESENT -- PROGRAM IS GOING");
           DISPLAY ("TO OPEN THE FILE WITH OUTPUT NEW");
           OPEN DISKFILE WITH OUTPUT NEW LOCK;
           CLOSE DISKFILE WITH LOCK;
           OPEN DISKFILE WITH INPUT;
         END;
CLOSE DISKFILE WITH REMOVE;

OPEN DISKFILE WITH OUTPUT LOCK_OUT;

CLOSE DISKFILE WITH REMOVE;

ZIP "RO OPEN;RE DISKFILE;";     % Resets the MCP OPEN option and
                                % removes DISKFILE.
STOP;

FINI;

% This example program shows various uses of the OPEN verb.
```

Output from Example Program:

```
OPENO =7275 BOJ. PP=4, MP=4 TIME = 15:37:20.3
OPEN=1
% OPENO =7275 FILE DISKFILE NOT PRESENT -- PROGRAM IS GOING
% OPENO =7275 TO OPEN THE FILE WITH OUTPUT NEW
OPENO =7275 "DISKFILE" OPENED SERIAL EXTEND OUTPUT NEW LOCK DISK
OPENO =7275 "DISKFILE" OPENED SERIAL EXTEND INPUT DISK
OPENO =7275 "DISKFILE" OPENED SERIAL EXTEND OUTPUT LOCKOUT
OPEN=0
"DISKFILE" REMOVED
OPENO =7275 EOJ. TIME = 15:37:41.6
```

OVERLAY

The OVERLAY verb is for MCP use only.

SDL Syntax:

```
—— OVERLAY (<interpreter-index>); ——————————————————————————|
```

Syntax Semantics:

interpreter-index
This field can be any valid SDL literal, identifier, or expression that returns a value and is used as an index by the interpreter swapper for the interpreter dictionary. The interpreter dictionary entry specifies the action that is to be taken.

Example:

```
OVERLAY (INDEX);
```

PARITY_ADDRESS

The PARITY_ADDRESS verb returns a 24-bit value which is the address of the first parity error in S-memory. If no parity error is encountered, the value @FFFFFF@ is returned. The PARITY_ADDRESS verb is used only by the MCP or by a standalone SDL program that does not run with the MCP. If the PARITY_ADDRESS verb is performed when the MCP is running, the MCP terminates the program.

SDL Syntax:

```
—— PARITY_ADDRESS ——————————————————————————————————————————————|
```

Example:

```
DECLARE BAD_ADDRESS BIT (24);         % The identifier BAD_ADDRESS is
BAD_ADDRESS := PARITY_ADDRESS;        % assigned the address of the
                                      % parity error.
```

PREVIOUS_ITEM

The PREVIOUS_ITEM verb causes the length field of the descriptor represented by <identifier> to be subtracted from the address field of that descriptor. This modified descriptor is put back onto the name stack and is also moved to the top of the evaluation stack. The modified descriptor that has been moved is the address of the new item described by < identifier >.

SDL Syntax:

—— PREVIOUS_ITEM (<identifier>) ——————————————————————————|

Syntax Semantics:

identifier

This field can be any valid SDL simple identifier.

Example:

```
DECLARE 01 CHAR_STRING   CHARACTER (1000),   % Causes the character D
           03 FILLER     CHARACTER (999),    % to be moved into the
           03 LAST_CHAR  CHARACTER (1);      % character immediately
PREVIOUS_ITEM (LAST_CHAR) := "D";            % prior to LAST_CHAR in
                                             % CHAR_STRING.
```

PROCESSOR_TIME

The PROCESSOR_TIME verb returns a 20-bit value that is the accumulated processor (CPU) time since beginning of job (BOJ). The time is returned in tenths of a second.

SDL and UPL Syntax:

```
—— PROCESSOR_TIME ——————————————————————————————————————————|
```

Example:

```
DECLARE X   BIT (24);
X := PROCESSOR_TIME;      % Assigns the 20-bit accumulated processor
                          % time into the identifier X.
```

Example Program:

```
DECLARE   HOURS          CHARACTER (2),
          MINUTES        CHARACTER (2),
          SECONDS        CHARACTER (2),
          TENTHS         CHARACTER (1),
          PROC_TIME      FIXED,
          X              FIXED,
          COUNTER        FIXED;

COUNTER := 0;
DO FOREVER;
   X := 9999999 * 9999999;
   IF ((BUMP COUNTER) > 900000) THEN UNDO;
END;

PROC_TIME := PROCESSOR_TIME;

HOURS   := SUBSTR (CONVERT ((PROC_TIME / 36000), CHARACTER), 6);
MINUTES := SUBSTR (CONVERT ((PROC_TIME MOD 36000 / 600), CHARACTER),6);
SECONDS := SUBSTR (CONVERT ((PROC_TIME MOD 600 / 10), CHARACTER), 6);
TENTHS  := SUBSTR (CONVERT ((PROC_TIME MOD 10), CHARACTER), 7);

DISPLAY ("THE TOTAL CPU TIME EQUALS " CAT HOURS CAT ":"
         CAT MINUTES CAT ":" CAT SECONDS CAT "." CAT TENTHS);
DISPLAY ("GOOD BYE");
STOP;
FINI;

% This example program multiplies two numbers 900,000 times and then
% uses the PROCESSOR_TIME verb to interrogate the CPU time.  The
% CPU time is then displayed on the ODT and the program goes to
% end of job.
```

PROGRAM_SWITCHES

The PROGRAM_SWITCHES verb returns the current values of the program switches from the program parameter block (PPB). If <switch-number> is specified, the 4-bit value of the specified program switch is returned. If <switch-number> is not specified, the 40-bit value of all 10 program switches is returned.

If <switch-number> contains a value which is less than zero or greater than nine, a run-time error results.

The program switches can be permanently set in the SDL/UPL program by using the MCP MODIFY command or set at run-time by using the MCP SWITCH program-attribute command. In either case, the program parameter block (PPB) for the SDL/UPL program contains the resulting value of the program switches.

The following shows how to modify the program switches in an SDL/UPL program at execution time.

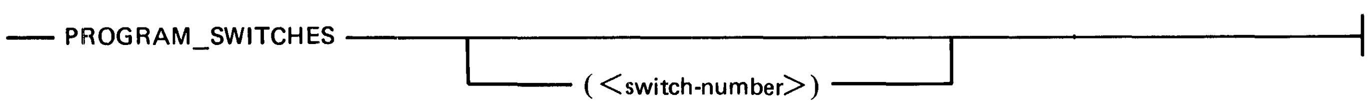
MODIFY <program name> SWITCH = @<value-0><value-1> ... <value-9>@
or
MODIFY <program name> SWITCH <switch number> = @<value>@

The following shows how to permanently modify the program switches in an SDL/UPL program.

EXECUTE <program name> SWITCH = @<value-0><value-2> ... <value-9>@
or
EXECUTE <program name> SWITCH <switch number> = @<value>@

Refer to the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982, for a complete description of the program switch attributes.

SDL and UPL Syntax:



Syntax Semantics:

switch-number
This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value. <switch-number> must have a value between 0 and 9, inclusive.

PROGRAM_SWITCHES

Examples:

```
X := PROGRAM_SWITCHES;              % Assigns to identifier X a 40-bit
                                    % value of all 10 program switches.

X := PROGRAM_SWITCHES (5);          % Assigns to identifier X a 4-bit
                                    % value of program switch 5.

X := PROGRAM_SWITCHES (Y);          % Assigns to identifier X a 4-bit
                                    % value of the program switch
                                    % specified by identifier Y.

X := PROGRAM_SWITCHES (BUMP Y);     % Assigns to identifier X a 4-bit
                                    % value of the program switch
                                    % specified by the value of Y + 1.
```

Example Program:

```
DECLARE    SWITCHES   BIT (40),
           INDEX      FIXED;

INDEX := 0;

SWITCHES := PROGRAM_SWITCHES;

DO FOREVER;
   DISPLAY ("SWITCH " CAT SUBSTR (CONVERT (INDEX, CHARACTER), 7)
            CAT " EQUALS " CAT
            CONVERT (SUBBIT (SWITCHES, (INDEX * 4), 4), CHARACTER));
   IF ((BUMP INDEX) > 9) THEN DO;
                                        DISPLAY ("GOOD BYE");
                                        STOP;
                                      END;
END;

FINI;

% This example program displays on the ODT the values of each
% program switch.  The PROGRAM_SWITCHES verb is used to interrogate
% the value of all ten switches.  The program switches must be set
% prior to or at execution time; otherwise, all the values are equal
% to @0@.
```

Output from Example Program:

```
?EXECUTE PRGSWITCHO SWITCH = @123456789A@;

PRGSWITCHO =7468 BOJ. PP=4, MP=4 TIME = 11:42:21.6
% PRGSWITCHO =7468 SWITCH 0 EQUALS 1
% PRGSWITCHO =7468 SWITCH 1 EQUALS 2
% PRGSWITCHO =7468 SWITCH 2 EQUALS 3
% PRGSWITCHO =7468 SWITCH 3 EQUALS 4
% PRGSWITCHO =7468 SWITCH 4 EQUALS 5
% PRGSWITCHO =7468 SWITCH 5 EQUALS 6
% PRGSWITCHO =7468 SWITCH 6 EQUALS 7
% PRGSWITCHO =7468 SWITCH 7 EQUALS 8
% PRGSWITCHO =7468 SWITCH 8 EQUALS 9
% PRGSWITCHO =7468 SWITCH 9 EQUALS A
% PRGSWITCHO =7468 GOOD BYE
PRGSWITCHO =7468 EOJ. TIME = 11:42:43.2
```

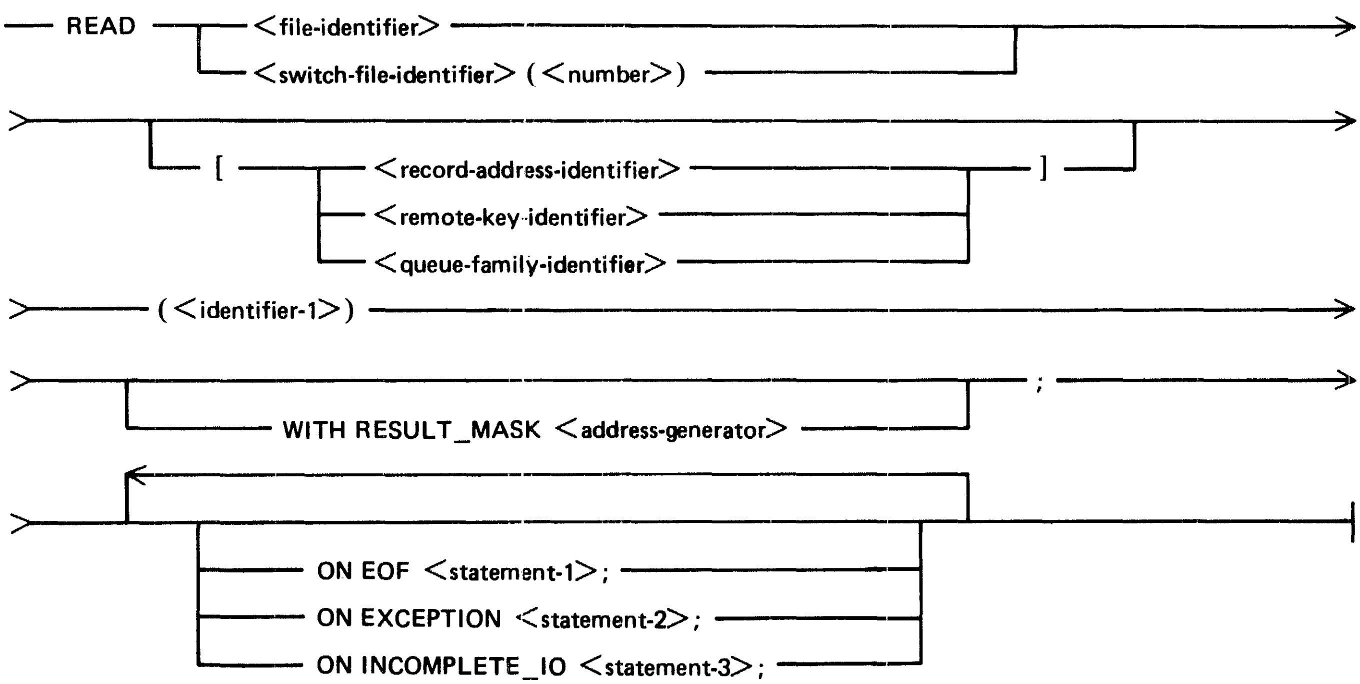
READ

The READ verb causes the SDL/UPL program to read a record from the specified file and store the record in <identifier-1>.

Read operations can be performed on any readable device. Reading a diskette file requires that the file be copied to a disk file before it is processed.

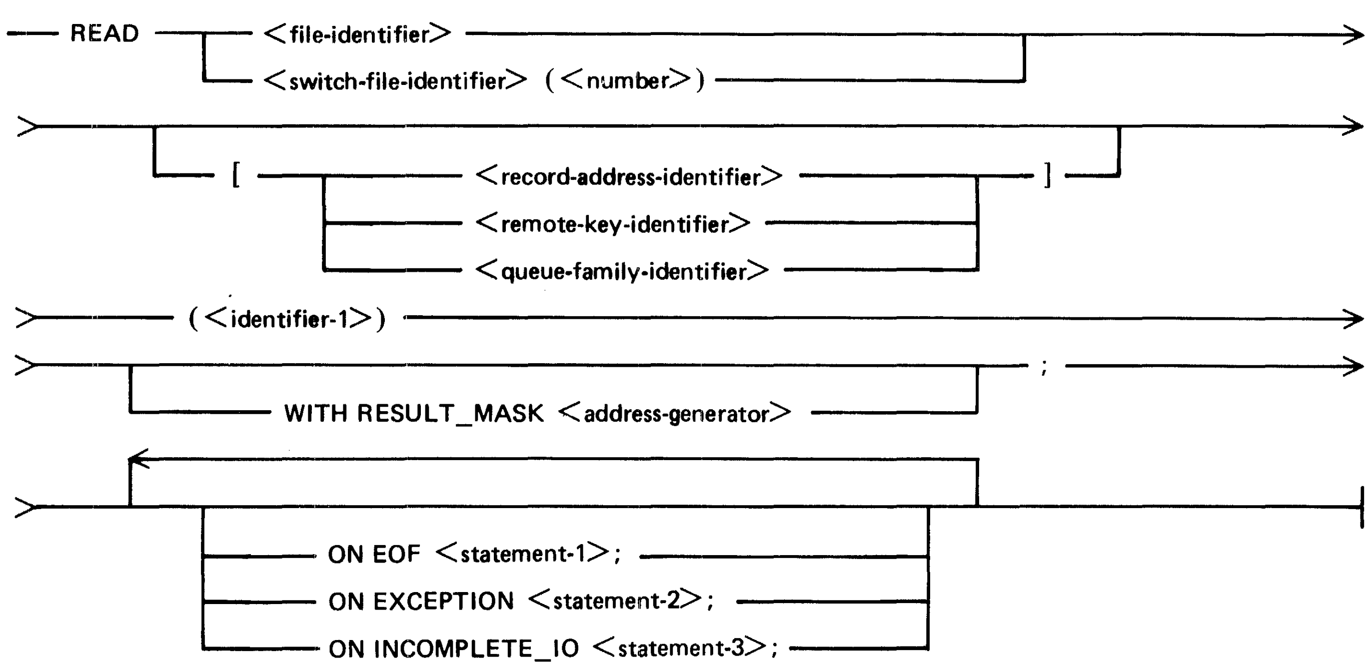
The file attributes in the FILE declaration statement determine which of the position options (<record-address-identifier>, <remote-key-identifier>, or <queue-family-identifier>) can be specified. The <record-address-identifier> requires a file with a disk device type and random access or a card device type with the STACKERS open attribute specified at file open time. The <remote-key-identifier> requires a file with a device type equal to REMOTE. The <queue-family-identifier> requires two file attributes to be specified in the file declaration. The two file attributes are a device type equal to QUEUE and the QUEUE_FAMILY_SIZE that is equal to the number of queue families.

SDL and UPL Syntax:



READ

UPL Syntax:



Syntax Semantics:

address-generator
: This field can be any valid SDL/UPL address generator. It specifies the name of the exception mask field.

file-identifier
: This field can be any valid SDL/UPL file identifier with exception of a file that is opened OUTPUT only and specifies the name of the file to be read.

switch-file-identifier
: This field can be any valid SDL/UPL switch file identifier with exception of a file that is opened OUTPUT only and specifies the name of the file to be read.

number
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value and specifies the file number of <switch-file-identifier>.

record-address-identifier
: This field can be any valid SDL/UPL identifier and it specifies the key location of a record within a file. <record-address-identifier> is valid for files with a device type equal to DISK RANDOM and DISK_PACK RANDOM. <record-address-identifier> is also valid for card files that are opened with the STACKERS open attribute.

<record-address-identifier> must be a binary value or an expression that returns a binary value. If the value is greater than 24 bits, only the rightmost 24 bits are used. For card files, the binary value of <record-address-identifier> must be less than or equal to seven, and must correspond to a stacker available on the device. For example, if only two stackers are available on the card device, a <record-address-identifier> equal to three is not valid.

READ

remote-key-identifier

This field can be any valid SDL/UPL identifier and it specifies the relative station number (RSN) within the remote file on which the READ operation is completed.

<remote-key-identifier> is valid for files with a device type equal to REMOTE. The data type of <remote-key-identifier> must be equal to CHARACTER and have a length of 10 bytes. A read operation of a remote file causes the relative station number of a station within the remote file, message text size and the read operation code "000" to be stored into <remote-key-identifier>. The relative station number defaults to the character "1" if the maximum number of stations in the remote file is equal to one. The maximum number of stations is specified in the FILE declarations. For example, DEVICE = REMOTE (5) specifies that the maximum number of stations for this file is five.

queue-family-identifier

This field can be any valid SDL/UPL identifier and it specifies the family number in the queue file which the read operation has completed.

<queue-family-identifier> is valid for a file with a device type equal to QUEUE and with the QUEUE_FAMILY_SIZE greater than one. <queue-family-identifier> specifies which queue family member from which to read. If <queue-family-identifier> is not specified in the READ verb, the oldest message in the queue file is read.

The end-of-file (EOF) record is treated as a pseudo-message in the queue file. That is, when the last message has been read from the queue file, the queue file remains not empty for waiting purposes. A subsequent read operation causes the end-of-file branch to be taken. The queue file is then empty but still in end-of-file status. If another read operation is issued to the queue file, the program takes the end-of-file branch. If the reading program closes and reopens the queue file or a new writing program opens the queue file, the end-of-file condition is reset.

A read operation directed to a specific member of a queue file family is treated as though it were issued to a simple queue file. A read operation issued to an unspecified member of a queue file family (unspecific read using <queue-family-identifier> equal to −1) returns the end-of-file condition if all the members in the queue file family are empty and no active writing programs have the queue file open.

identifier-1

This field can be any valid SDL/UPL identifier and it specifies the data address in which to store the data read.

ON EOF

The keywords ON EOF cause the program to perform <statement-1>, if the end-of-file record is read from the file. For queue files, if end of file occurs, the queue file is then empty and there are no programs with the file opened and the OUTPUT open attribute set.

ON EXCEPTION

The keywords ON EXCEPTION cause the program to either perform <statement-2> or to store the 24-bit exception mask into <identifier-2>. If a parity error is encountered during the read operation and all the MCP retries have been exhausted, the 24-bit exception is stored in <identifier-2>.

Exceptions for a file can be masked if the EXCEPTION_MASK file attribute is specified in the FILE declaration statement. If an identifier, enclosed in parentheses, follows the ON EXCEPTION keywords, a 24-bit value which describes the exception that occurred is returned.

ON INCOMPLETE_IO
The key words ON INCOMPLETE_IO cause the program to perform <statement-3>, if the queue file is empty and another program has opened the queue file with the OUTPUT open attribute set.

statement-1
This field can be any valid SDL/UPL statement. It is performed when the ON EOF keywords are specified in the READ verb and the end-of-file record is encountered in the file. If an exception occurs for queue files, an invalid <remote-key-identifier> value has been provided in the READ verb.

statement-2
This field can be any valid SDL/UPL statement. It is performed when the ON EXCEPTION keywords are specified in the READ statement and a parity error is encountered while attempting to read a record from the file.

statement-3
This field can be any valid SDL/UPL statement. It is performed when the ON INCOMPLETE_IO keywords are specified in the READ statement, when the end-of-file record was encountered in the queue file, and when there is a program that has the queue file open with the OUTPUT open attribute.

WITH RESULT_MASK
The keywords WITH RESULT_MASK cause the program to use <address-generator> as the exception mask identifier. The EXCEPTION_MASK file attribute must be specified in its FILE declaration statement.

Variable-Length Records

The syntax of the READ verb for variable-length records resembles the syntax for fixed-length records. The difference between them is the data type and the data length of the identifier.

Variable-length records are allowed only in tape and serial disk files that are declared with the file attribute VARIABLE. The RECORDS file attribute of the file must be large enough to hold the largest record that is to be read or written.

The actual manipulation of variable-length records is invisible to the programmer of the read operation. An exception is that the programmer must allow for a 4-byte field, which begins in the first position of each record to be stored in the identifier receiving the data. This 4-byte character field contains the length, in bytes, of the record that is read. This record length is equal to the number of bytes in the data file plus four. The record length is specified as a decimal value.

READ

Example Program that Reads Variable-Length Records:

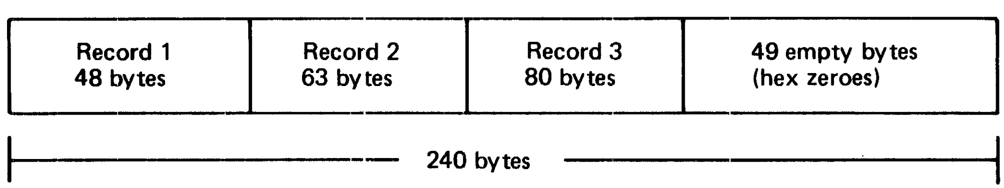
```
FILE PAYROLL (DEVICE = DISK,
              OPEN_OPTION = INPUT/OUTPUT,
              RECORDS = 240/1, VARIABLE);

DECLARE    01 DISK_BUFFER   CHARACTER (80),
              02 REC_SIZE   CHARACTER (4),
              02 DATA       CHARACTER (76);

DO FOREVER;
   READ PAYROLL (DISK_BUFFER);
        ON EOF UNDO;
END;

CLOSE PAYROLL LOCK;
STOP;
FINI;
```

To process variable-length records, the MCP builds a single buffer whose size is equal to the declared record size multiplied by the records per block. The MCP reads into its buffer as many complete logical records as it can. It never splits a logical record across physical record boundaries.

The following shows those logical records read into the buffer by the MCP. Assume the program specifies a record size equal to 240 bytes and the order and length of each record are:

Record Number	Record Size in Bytes (Including Record Size Field)
1	48
2	63
3	80
4	
5	31

Figure 9-1 shows the contents of the 240-byte program buffer after a read operation is performed.



G18304

Figure 9-1. Contents of Buffer After a Read Operation.

Only records 1, 2, and 3 are stored into the buffer because the next record (record 4) is too long to be stored in the remaining portion of the buffer. The unused portion of the buffer is filled with hexadecimal zeroes.

Examples:

```
READ DISKFILE (FIELD);                              % Reads from the file
     ON EOF STOP;                                   % labeled DISKFILE.

READ DISK [INDEX] (FIELD);                          % Reads from the file
     ON EOF STOP;                                   % labeled DISK at
     ON EXCEPTION DISPLAY ("NOT FOUND");            % record address =
                                                    % the value of INDEX.

READ QUEUEFILE [NUMBER] (FIELD);                    % Reads from the file
     ON INCOMPLETE_IO DISPLAY ("NO MESSAGES");      % labeled QUEUEFILE
     ON EOF DISPLAY ("NO WRITERS");                 % at queue family =
     ON EXCEPTION DISPLAY ("INVALID KEY");          % the value of NUMBER.

READ REMOTEFILE [KEY] (FIELD);                      % Reads from the file
     ON EXCEPTION DISPLAY ("INVALID KEY");          % labeled REMOTEFILE
                                                    % at remote key = the
                                                    % value of KEY.
```

READ

Example Program:

```
DECLARE  FIELD  CHARACTER (90);
FILE DISKFILE (DEVICE = DISK,
               RECORDS = 90/2);
OPEN DISKFILE WITH INPUT;
DO FOREVER;
   CASE WAIT (TIME_TENTHS (10), SPO_INPUT_PRESENT);
   % TIME = 1 SECOND
     DO;
        READ DISKFILE (FIELD);
           ON EOF DO;
                    DISPLAY ("END OF FILE ENCOUNTERED -- GOOD BYE");
                    STOP;
                 END;
           ON EXCEPTION DO;
                          DISPLAY ("PARITY ENCOUNTERED -- GOOD BYE");
                          STOP;
                       END;
        DISPLAY (FIELD);
     END;
   % SPO_INPUT_PRESENT
     DO;
        ACCEPT FIELD;
        IF FIELD = "BYE" THEN DO;
                                 DISPLAY ("GOOD BYE");
                                 STOP;
                              END;
     END;
   END CASE;
END;
FINI;

% This example program reads a disk file labeled DISKFILE and
% displays on the ODT each record read.  If the end-of-file
% record is encountered or an exception occurs, the program
% goes to end of job.  If BYE is entered to the program, the
% program goes to end of job.
```

READ_CASSETTE

The READ_CASSETTE verb causes the number of bits specified by <destination-identifier> to read from the console cassette drive to the address specified by that <destination-identifier>. This number of bits must be equal to the record size minus the hash-total size (if it is present) of 16 bits. The keywords HASH_TOTAL or NO_HASH_TOTAL indicate whether or not a hash-total is expected at the end of the record.

SDL Syntax:

```
── READ_CASSETTE (<destination-identifier>, ──────────┬── HASH_TOTAL ─────┬──────>
                                                      └── NO_HASH_TOTAL ──┘

>────── , <result-identifier>); ─────────────────────────────────────────────────┤
```

Semantics:

destination-identifier
: This field specifies the number of bits to be read from the console cassette drive and specifies the destination field for the data.

result-identifier
: This field contains a value of 0 or 1 after the READ_CASSETTE operation is complete. A value of 0 indicates that the hash total was incorrect. A value of 1 indicates that the hash total was correct.

HASH_TOTAL
: The keyword HASH_TOTAL specifies that a hash total is expected at the end of the record.

NO_HASH_TOTAL
: The keyword NO_HASH_TOTAL specifies that there is no hash total expected at the end of the record.

Examples:

```
READ_CASSETTE (DESTINATION, HASH_TOTAL, RESULT);

READ_CASSETTE (RECORD, NO_HASH_TOTAL, HASH_RESULT);
```

READ_CASSETTE

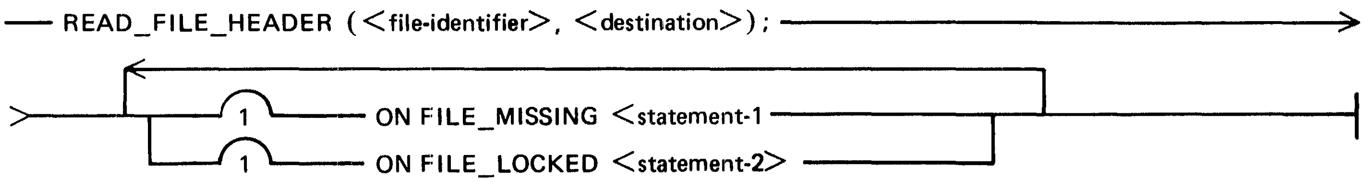
Example Program:

```
FILE LINE (DEVICE = PRINTER,
           RECORDS = 132/1);

DECLARE  CASSETTE_RECORD  BIT (80),
         HASH_RESULT      BIT (1);

OPEN LINE OUTPUT NEW;
DO FOREVER;
   READ_CASSETTE (CASSETTE_RECORD, HASH_TOTAL, HASH_RESULT);
   IF HASH_RESULT = 1
      THEN WRITE LINE (CONVERT (CASSETTE_RECORD, CHARACTER));
      ELSE DO;
              DISPLAY "INCORRECT HASH RESULT";
              CLOSE LINE;
              STOP;
           END;
END;
FINI;

% This example program reads from the console cassette drive
% using the READ_CASSETTE verb and writes the data to a printer
% file labeled LINE.
```

READ__FILE__HEADER

The READ__FILE__HEADER verb reads the disk-file-header information for the file specified by <file-identifier>. This verb is intended for use only in B 1000 system software.

SDL Syntax:



Syntax Semantics:

file-identifier

This field specifies the name of the file and can be any valid SDL literal, identifier, or expression that returns a value with a data type equal to CHARACTER. <file-identifier> is expected to be a 30-character value, where the first 10 characters are the pack identifier, the second 10 characters are the multifile identifier, and the third 10 characters are the file identifier. Each file identifier is left-justified in its respective field. If only one file name exists (no multifile identifier or pack identifier), the file name is left-justified in the second 10 characters of <file-identifier> and the first and third 10 characters are set to blank.

destination

This field can be any valid SDL identifier and it specifies the receiving field for the disk-file-header information. This field is expected to be from 576 to 4320 bits in length depending upon the number of disk areas allocated for the file.

ON FILE__MISSING

The keywords ON FILE__MISSING cause <statement-1> to be performed if the file name specified by <file-identifier> is not in the disk directory.

ON FILE__LOCKED

The keywords ON FILE__LOCKED cause <statement-2> to be performed if the file name specified by <file-identifier> is opened by another program with the LOCK open option set.

statement-1

This field can be any valid SDL statement and it is performed if the keywords ON FILE__MISSING are specified and <file-identifier> is not in the disk directory.

statement-2

This field can be any valid SDL statement and it is performed if the keywords ON FILE__LOCKED are specified and <file-identifier> is currently opened with the LOCK open option set.

READ_FILE_HEADER

Example:

```
DECLARE DISKFILE      CHARACTER (30),                    % The disk file header
        DESTINATION   BIT (4320);                        % information of the file
DISKFILE := "USER       MASTER     FILE            ";    % USER/MASTER/FILE is
READ_FILE_HEADER (DISKFILE, DESTINATION);                % stored in DESTINATION.
 ON FILE_MISSING STOP;
 ON FILE_LOCKED STOP;
```

Example Program:

```
& VSSIZE 80000
& NSSIZE 40
 DECLARE FILENAME      CHARACTER (30),
         DESTINATION   BIT (4320),
         DFH_LENGTH    BIT (16);

 DO MAIN_LOOP FOREVER;
    DISPLAY ("ENTER THE 30 CHARACTER FILE NAME LEFT JUSTIFIED OR ENTER "
             CAT "BYE TO GO TO EOJ");
    ACCEPT FILENAME;
    IF FILENAME = "BYE" THEN DO;
                                DISPLAY ("GOOD BYE");
                                STOP;
                             END;
    DO READ_DFH;

       READ_FILE_HEADER (FILENAME, DESTINATION);
         ON FILE_MISSING DO;
                            DISPLAY ("FILE " CAT FILENAME CAT
                                     "NOT IS DISK DIRECTORY");
                            UNDO READ_DFH;
                         END;
         ON FILE_LOCKED DO;
                           DISPLAY ("FILE " CAT FILENAME CAT
                                    " IS LOCKED");
                           UNDO READ_DFH;
                        END;

      DFH_LENGTH := SUBBIT (DESTINATION, 91, 16);
      DISPLAY (" THE DISK FILE HEADER OF " CAT FILENAME CAT " IS");
      DISPLAY (CONVERT (SUBBIT(DESTINATION, 0, DFH_LENGTH), CHARACTER));
    END READ_DFH;
 END MAIN_LOOP;
 FINI;

 % This example program displays the disk-file-header information
 % for the file name that is accepted from the ODT.  If BYE is
 % entered, the program goes to end of job.
```

READ__FPB

The READ__FPB verb reads the file parameter block (FPB) of the file specified by <file-identifier> or <file-number> and stores the information in <destination>.

SDL Syntax:

```
── READ_FPB ( ──┬── <file-identifier> ──┬── , <destination>) ; ──────────┤
                └── <file-number> ──────┘
```

Syntax Semantics:

file-identifier
: This field can be any valid SDL file identifier and it specifies the file name from which to read the file parameter block (FPB) information.

file-number
: This field can be any valid SDL number and it specifies the relative file number, within the program, from which to read the file parameter block (FPB) information. The relative file numbers range from 0 to n-1, where n is the total number of files declared in the SDL program.

destination
: This field can be any valid SDL identifier and it specifies the (FPB) information. The length of this field must be 2096 bits.

Example:

```
DECLARE FPB_INFO  BIT (1440);      % The file parameter block information
READ_FPB (DISKFILE, FPB_INFO);     % of the file DISKFILE is stored into
                                   % identifier FPB_INFO.
```

READ_FPB

Example Program:

```
DECLARE ODT_INPUT  CHARACTER (10),
        01 FPB_RECORD      BIT (1440),
           03 FILE_NAME    CHARACTER (10);

FILE DISKFILE  (DEVICE = DISK,
                RECORDS = 180/10);

OPEN DISKFILE WITH OUTPUT NEW;
DO FOREVER;

   READ_FPB (DISKFILE, FPB_RECORD);

   DISPLAY ("THE FPB NAME OF DISKFILE IS " CAT FILE_NAME );
   DISPLAY ("ENTER ANY 10-CHARACTER FOR THE NEW NAME OF DISKFILE"
            CAT " OR ENTER BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                             DISPLAY ("GOOD BYE");
                             CLOSE DISKFILE WITH RELEASE;
                             STOP;
                          END;
   FILE_NAME := ODT_INPUT;

   WRITE_FPB (DISKFILE, FPB_RECORD);

END;
FINI;

% This example program uses the READ_FPB to read the file parameter
% block information from the file DISKFILE and uses the WRITE_FPB
% verb to change the name of file DISKFILE.  The program displays
% the current file name that is currently stored in the file
% parameter block, accepts a 10-character file name, and stores the
% new file name in the file parameter block.  If BYE is entered, the
% program goes to end of job.
```

READ__OVERLAY

The READ__OVERLAY verb reads from the disk address specified in <overlay-information> and stores the beginning and ending addresses and disk address of the data segment.

The READ__OVERLAY verb is used only by the SDL intrinsics.

SDL Syntax:

```
—— READ_OVERLAY ( <overlay-information> ) ; ——————————————————————————————|
```

Syntax Semantics:

overlay-information
This field can be any valid SDL literal, identifier, or expression that returns a 76-bit value and has the following format.

Bits	Description
0-3	EU = 0 (not used).
4-27	Base-relative beginning address.
28-51	Base-relative ending address.
52-75	Disk address, relative to program area.

Example:

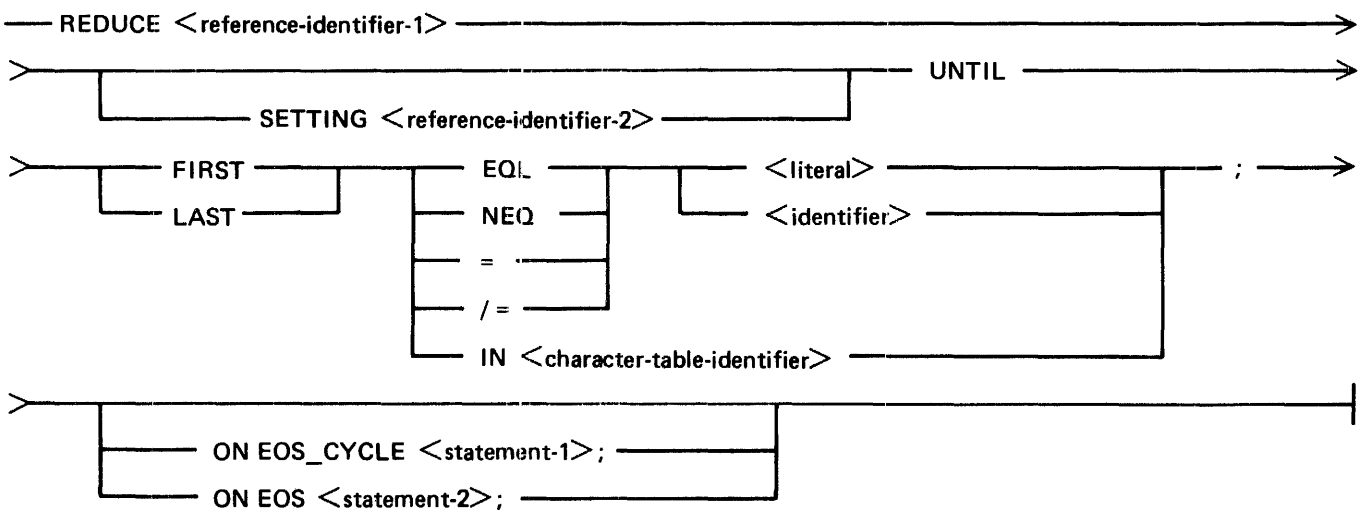
```
DECLARE 01 OVERLAY_RECORD  BIT (76),   % The data segment at disk
           03 EU           BIT (4),    % address @008A78@ is stored
           03 BEGIN_ADDR   BIT (24),   % in the base to limit of the
           03 END_ADDR     BIT (24),   % program beginning at @71F7A2@
           03 DISK_ADDR    BIT (24);   % and ending at @71F842@.
EU := 1;
BEGIN_ADDR := @71E7A2@;
END_ADDR := @71F842@;
DISK_ADDR := @008A78@;
READ_OVERLAY (OVERLAY_RECORD);
```

REDUCE

The REDUCE verb truncates a reference identifier from the left (right) until the first (last) character satisfies a specified condition. This is a flexible and efficient means for scanning character strings which use reference variables, rather than integers which serve as pointers to substrings.

No change is actually made to the value of an identifier when the REDUCE verb is performed. The identifier is re-bound to a substring of its former reference identifier.

SDL and UPL Syntax:



Syntax Semantics:

reference-identifier-1
: This field can be any valid SDL/UPL reference identifier and it specifies the reference variable to be reduced.

reference-identifier-2
: This field can be any valid SDL/UPL reference identifier and it specifies the reference variable that contains the truncated portion of <reference-identifier-1>. <reference-identifier-2> is assigned the truncated portion of <reference-identifier-1> when the keyword SETTING is specified.

SETTING
: The keyword SETTING causes the truncated portion of <reference-identifier-1> to be stored in <reference-identifier-2>.

UNTIL
: The keyword UNTIL is required.

FIRST
: The keyword FIRST causes the reduction to end on the first character that is equal or not equal to the specified literal or identifier or in the specified <character-identifier-table>.

LAST
: The keyword LAST causes the reduction to end on the last character that is equal or not equal to the specified literal or identifier or in the specified <character-identifier-table>.

EQL

The keyword EQL specifies that the reduction is complete when a character in <reference-identifier-1> is equal to the specified literal or identifier.

NEQ

The keyword NEQ specifies that the reduction is complete when a character in <reference-identifier-1> is not equal to the specified literal or identifier.

=

The keysymbol = has the same meaning as the EQL keyword.

/=

The keysymbols /= have the same meaning as the NEQ keyword.

IN

The keyword IN specifies that the reduction is complete when a character in <reference-identifier-1> is in the character table specified by <character-table-identifier>.

literal

This field can be any valid SDL/UPL literal and it specifies the character within <reference-identifier-1> that ends the reduction. This character must be enclosed within the quotation mark (") characters.

identifier

This field can be any valid SDL/UPL 1-character identifier and it specifies the character within <reference-identifier-1> that ends the reduction.

character-table-identifier

The field can be any valid character table identifier and it specifies the characters within <reference-identifier-1> that ends the reduction.

ON EOS

The keywords ON EOS cause <statement-2> to be performed. Control is returned to the statement that follows the REDUCE verb if <reference-identifier-1> is reset and no longer null. <reference-identifier-1> can become null when the reduction ends with <reference-identifier-1> equal to "".

ON EOS_CYCLE

The keywords ON EOS_CYCLE cause <statement-1> to be performed. Control is returned to the REDUCE verb if <reference-identifier-1> is reset and is no longer null. <reference-identifier-1> can become null when the reduction ends with the reference identifier equal to "".

Example 1:

```
DECLARE IDENTIFIER     CHARACTER (6),
        REFERENCE_ID   REFERENCE;

IDENTIFIER := "ABCDEF";

REFER REFERENCE_ID TO IDENTIFIER;

REDUCE REFERENCE_ID UNTIL FIRST ="D";
```

REDUCE

Figure 9-2 shows the before and after results of example 1.

REFERENCE_ID Before				REFERENCE_ID After
"ABCDEF"	——>	REDUCE	——>	"DEF"

G18305

Figure 9-2. Before and After Results of the REDUCE Operation

The truncated portion of the string can also be referenced by using the SETTING keyword in the REDUCE verb.

Example 2:

```
DECLARE IDENTIFIER        CHARACTER (6),
        REFERENCE_ID_1    REFERENCE,
        REFERENCE_ID_2    REFERENCE;

IDENTIFIER := "ABCDEF";

REFER REFERENCE_ID_1 TO IDENTIFIER;

REDUCE REFERENCE_ID_1 SETTING REFERENCE_ID_2 UNTIL FIRST ="D";
```

Figure 9-3 shows the before and after results of example 2.

REFERENCE_ID_1 Before				REFERENCE_ID_1 After	REFERENCE_ID_2 After
"ABCDEF"	——>	REDUCE	——>	"DEF"	"ABC"

G18306

Figure 9-3. Before and After Results of the REDUCE Operation

The reduction of an identifier can also be performed from right to left by using the keyword LAST instead of the keyword FIRST.

Example 3:

```
DECLARE IDENTIFIER        CHARACTER (6),
        REFERENCE_ID_1    REFERENCE,
        REFERENCE_ID_2    REFERENCE;

IDENTIFIER := "ABCDEF";

REFER REFERENCE_ID_1 TO IDENTIFIER;

REDUCE REFERENCE_ID_1 SETTING REFERENCE_ID_2 UNTIL LAST ="D";
```

Figure 9-4 shows the before and after results of the example 3.

REFERENCE_ID_1 Before				REFERENCE_ID_1 After	REFERENCE_ID_2 After
"ABCDEF"	⟶	REDUCE	⟶	"ABCD"	"EF"

G18307

Figure 9-4. Before and After Results of the REDUCE Operation

Example Program:

```
DECLARE  ODT_INPUT      CHARACTER (50),
         REFERENCE_1    REFERENCE,
         REFERENCE_2    REFERENCE;

DO FOREVER;
   DISPLAY ("ENTER ANY 50 CHARACTERS OR ENTER BYE TO GO TO EOJ");
   ACCEPT ODT_INPUT;
   REFER REFERENCE_1 TO ODT_INPUT;

   REDUCE REFERENCE_1 UNTIL FIRST NEQ " ";
     ON EOS DO;
               DISPLAY ("NO CHARACTERS WERE ENTERED - BYE ASSUMED");
               DISPLAY ("GOOD BYE");
               STOP;
            END;

   REDUCE REFERENCE_1 SETTING REFERENCE_2 UNTIL FIRST EQL " ";

   IF REFERENCE_2 = "BYE" THEN DO;
                                    DISPLAY ("GOOD BYE");
                                    STOP;
                                 END;
   DISPLAY ("THE FIRST NON-BLANK WORD IS " CAT REFERENCE_2);
END;

FINI;

% This example program accepts up to 50 characters on the ODT
% and uses the REDUCE verb to scan for the first group of
% characters delimited by the blank character.  The REFER verb
% is used to bind REFERENCE_1 to ODT_INPUT.
```

REFER

The REFER verb binds a reference identifier to an addressable data item. It then becomes the referent of the reference identifier.

The lexic level of the identifier cannot be greater than that of <reference-identifier>.

A reference identifier can be bound to a NULL character or a bit string. Testing for NULL is accomplished by examining the reference identifier for a length of 0 (zero).

SDL and UPL Syntax:

```
—— REFER <reference-identifier> TO <identifier>; ——————————————————————|
```

Syntax Semantics:

reference-identifier
: This field must be an identifier with a data type equal to REFERENCE.

identifier
: This field can be any valid SDL/UPL identifier and it specifies the data item that is to be bound to <reference-identifier>.

Examples:

```
DECLARE    CHAR_ID          CHARACTER (20),
           REFER_CHAR_ID    REFERENCE,
           BIT_ID           BIT (20),
           REFER_BIT_ID     REFERENCE;

REFER REFER_CHAR_ID TO CHAR_ID;       % REFER_CHAR_ID is now bound to
                                      % CHAR_ID.

REFER REFER_BIT_ID TO BIT_ID;         % REFER_BIT_ID is now bound to
                                      % BIT_ID.
```

Example Program:

For an example program using the REFER verb, refer to the three REDUCE verb programs.

REFER_ADDRESS

The REFER_ADDRESS verb causes the base-relative address of <address> to be stored in the address part of <reference-identifier>.

SDL Syntax:

```
── REFER_ADDRESS (<reference-identifier>, <address>); ──────────────────┤
```

Syntax Semantics:

reference-identifier
: This field can be any valid SDL reference identifier and it specifies the field that will receive the base-relative address of <address>.

address
: This field can be any valid SDL literal, identifier, or expression that returns a value. The address of <address> is stored in the address part of <reference-identifier>.

Example:

```
DECLARE REF     REFERENCE,          % The value of identifier A is
        A       CHARACTER (10),     % stored in the address part of
REFER_ADDRESS (REF, A);             % reference identifier REF.
```

Example Program:

```
RECORD  R
        R_A       BIT (80);
DECLARE ADDRESS R REFERENCE,
        A       R;

REFER_ADDRESS (ADDRESS, DATA_ADDRESS (A));
DISPLAY ("THE DATA ADDRESS OF IDENTIFIER A IS " CAT
          CONVERT (DATA_ADDRESS (A), CHARACTER));
DISPLAY ("THE DATA ADDRESS OF REFERENCE IDENTIFIER ADDRESS IS " CAT
          CONVERT (DATA_ADDRESS (ADDRESS), CHARACTER));
STOP;
FINI;

% This example program stores the address of record R into the
% address part of reference identifier ADDRESS and displays the
% address of each identifier.
```

Output from Example Program:

```
ADDRESSO =2159 BOJ. PP=4, MP=4 TIME = 15:42:09.3
% ADDRESSO =2159 THE DATA ADDRESS OF IDENTIFIER A IS 000000
% ADDRESSO =2159 THE DATA ADDRESS OF REFERENCE IDENTIFIER ADDRESS
   IS 000000
ADDRESSO =2159 EOJ. TIME = 15:42:16.1
```

REFER_LENGTH

The REFER_LENGTH verb causes the length of <length> to be stored in the length part of <reference-identifier>.

SDL Syntax:

```
—— REFER_LENGTH (<reference-identifier>, <length>) ; ————————————————————————|
```

Syntax Semantics:

reference-identifier

This field can be any valid SDL reference identifier and it specifies the field in which to receive <length>.

length

This field can be any valid SDL literal, identifier, or expression that returns a value. The length of <length> is stored in the length part of <reference-identifier>.

Example:

```
DECLARE REF     REFERENCE,          % The length of identifier LENGTH
        LENGTH FIXED;               % is stored in the length part of
REFER_LENGTH (REF, LENGTH);         % reference identifier REF.
```

Example Program:

```
DECLARE LENGTH REFERENCE,
        A      FIXED;

REFER_LENGTH (LENGTH, DATA_LENGTH (A));
DISPLAY ("THE DATA LENGTH OF IDENTIFIER A IS " CAT
         CONVERT (DATA_LENGTH (A), CHARACTER));
DISPLAY ("THE DATA LENGTH OF REFERENCE IDENTIFIER LENGTH IS " CAT
         CONVERT (DATA_LENGTH (LENGTH), CHARACTER));
STOP;
FINI;

% This example program stores the value of identifier LENGTH in
% the length part of reference identifier REF and displays the
% length of each identifier.
```

Output from Example Program:

```
LENGTH =2178 BOJ. PP=4, MP=4 TIME = 16:02:18.0
% LENGTH =2178 THE DATA LENGTH OF IDENTIFIER A IS 000018
% LENGTH =2178 THE DATA LENGTH OF REFERENCE IDENTIFIER LENGTH IS
  000018
LENGTH =2178 EOJ. TIME = 16:02:25.8
```

REFER__TYPE

The REFER__TYPE verb causes the data type of <type> to be stored in the data type part of <reference-identifier>.

SDL Syntax:

```
—— REFER_TYPE (<reference-identifier>, <type>); ——————————————————————————|
```

Syntax Semantics:

reference-identifier
: This field can be any valid SDL reference identifier and it specifies the field that will receive the data type.

type
: This field can be any valid SDL literal, identifier, or expression that returns a value. The data type of <type> is stored in the data type part of <reference-identifier>.

Example:

```
DECLARE REF     REFERENCE,          % The data type of identifier TYPE
        TYPE    BIT (5);            % is stored in the data type part
REFER_TYPE (REF, TYPE);             % of reference identifier REF.
```

Example Program:

```
DECLARE TYPE REFERENCE,
        A    FIXED;

REFER_TYPE (TYPE, DATA_TYPE (A));
DISPLAY ("THE DATA TYPE OF IDENTIFIER A IS " CAT
            CONVERT (DATA_TYPE (A), CHARACTER));
DISPLAY ("THE DATA TYPE OF REFERENCE IDENTIFIER TYPE IS " CAT
            CONVERT (DATA_TYPE (TYPE), CHARACTER));
STOP;
FINI;

% This example program stores the data type of identifier A in
% the data type part of reference identifier TYPE and displays
% the data type of each identifier.
```

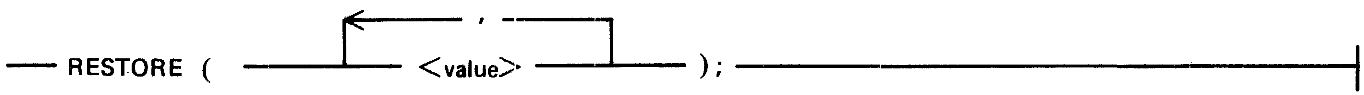
Output from Example Program:

```
TYPEO =2174 BOJ. PP=4, MP=4 TIME = 15:54:23.2
% TYPEO =2174 THE DATA TYPE OF IDENTIFIER A IS 000044
% TYPEO =2174 THE DATA TYPE OF REFERENCE IDENTIFIER TYPE IS 000044
TYPEO =2174 EOJ. TIME = 15:54:31.0
```

RESTORE

The RESTORE verb assigns an evaluation stack entry to each specified value, beginning with the top of the evaluation stack. This verb is used in conjunction with the SAVE verb.

SDL Syntax:



Syntax Semantics:

value
: This field can be any valid SDL identifier or expression that returns a value and specifies the value to be placed on the evaluation stack.

Example:

```
SAVE (A, B, C);
   •
   •
   •
RESTORE (C, B, A);
```

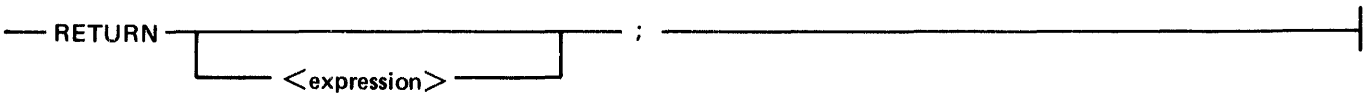
RETURN

The RETURN verb can take one of two forms, depending on the type of procedure encompassing it. If the procedure is a typed procedure, an expression must be returned to the point of invocation. If the procedure is a non-typed procedure, only a simple return is required.

Type checking on the RETURN verb is performed only at run time when the FORMAL_CHECK compiler option is specified as a compiler control option.

The SDL/UPL compiler generates an implicit RETURN verb if one is not specified and the RETURN verb is required. Refer to Section 7 for use of the RETURN verb.

SDL and UPL Syntax:



Syntax Semantics:

expression
    This field can be any valid SDL/UPL expression and it specifies the value that is to be returned to the point where the procedure was invoked.

Examples:

```
RETURN;

RETURN 1;
```

RETURN_AND_ENABLE_INTERRUPTS

The RETURN_AND_ENABLE_INTERRUPTS verb is used only by the MCP. This verb causes a normal procedure exit to occur and enables interrupt.

SDL Syntax:

```
—— RETURN_AND_ENABLE_INTERRUPTS: ——————————————————————————————————|
```

Example:

```
RETURN_AND_ENABLE_INTERRUPTS;
```

REVERSE_STORE

The REVERSE_STORE verb performs a number of assignment operations and is more efficient than separately specifying each assignment operation.

The REVERSE_STORE verb assigns each address generator and expression in the following order. <address-generator-1> is assigned the value of <address-generator-2>, <address-generator-2> is assigned the value of <address-generator-3>, ... , <address-generator-n-1> is assigned the value of <address-generator-n>, and <address-generator-n> is assigned the value of <expression>.

SDL and UPL Syntax:

```
── REVERSE_STORE (<address-generator-1>, <address-generator-2>, ──────────>

>──────<──────>, <address-generator-n>, <expression>) ──────────────────┤
```

Syntax Semantics:

address-generator-1 thru address-generator-n
: These fields can be any valid SDL/UPL address generators where n represents any number and it specifies the fields that perform the multiple assignment operations.

expression
: This field can be any valid SDL/UPL expression and it specifies the value to assign to <address-generator-n>.

Example 1:

```
REVERSE_STORE (A, B, "1");          % Identifier A is assigned the value
                                    % of identifier B, and identifier B
                                    % is assigned the character 1.
```

Example 2:

```
REVERSE_STORE (A, B, C, D+1);       % Identifier A is assigned the
                                    % value of identifier B, identifier
                                    % B is assigned the value of
                                    % identifier C, and identifier C is
                                    % assigned the value of identifier
                                    % D+1.
```

Example 3:

```
REVERSE_STORE                       % Identifier A is assigned the
 (A, IF 1 > Z THEN B ELSE C,        % value of either identifier B
  CASE V-1 OF (M, N, O), X-1);      % or C depending on the result
                                    % of evaluating the expression
                                    % 1 > Z.  Identifier B or C is
                                    % assigned the value of identifier
                                    % M, N, or O depending on the
                                    % result of evaluating the
                                    % expression V-1.  Identifier M, N,
                                    % or O is assigned the value of X-1.
```

REVERSE_STORE

Example Program:

```
DECLARE    A(9)       CHARACTER (10),
           COUNTER    FIXED,
           B_TIME     BIT (20),
           A_TIME     BIT (20);

COUNTER := 0;
B_TIME := PROCESSOR_TIME;
DO FOREVER;
  REVERSE_STORE (A(0), A(1), A(2), A(3), A(4), A(5), A(6), A(7), A(8));
  IF ((BUMP COUNTER) = 100000) THEN UNDO;
END;
A_TIME := PROCESSOR_TIME;
DISPLAY ("THE PROCESSOR TIME FOR PERFORMING 100000 REVERSE_STORE " CAT
         "OPERATIONS IS " CAT DECIMAL((A_TIME - B_TIME), 4) CAT
         " TENTHS OF SECONDS");

COUNTER := 0;
B_TIME := PROCESSOR_TIME;
DO FOREVER;
   A(0) := A(1);  A(1) := A(2);  A(2) := A(3);  A(3) := A(4);
   A(4) := A(5);  A(5) := A(6);  A(6) := A(7);  A(7) := A(8);
   IF ((BUMP COUNTER) = 100000) THEN UNDO;
END;
A_TIME := PROCESSOR_TIME;
DISPLAY ("THE PROCESSOR TIME FOR PERFORMING 100000 SEPARATE ASSIGNMENT"
        CAT " OPERATIONS IS " CAT DECIMAL((A_TIME - B_TIME), 4) CAT
         " TENTHS OF SECONDS");

STOP;
FINI;

% This example program compares the amount of processor time that is
% used for the REVERSE_STORE verb and the assignment operations in
% assigning the same amount of information.  The REVERSE_STORE is
% significantly more efficient.
```

Output from Example Program:

```
% REVERSED =1283 THE PROCESSOR TIME FOR PERFORMING 100000
      REVERSE_STORE OPERATIONS IS 0745 TENTHS OF SECONDS
% REVERSED =1283 THE PROCESSOR TIME FOR PERFORMING 100000
      SEPARATE ASSIGNMENT OPERATIONS IS 0981 TENTHS OF SECONDS
```

SAVE

The SAVE verb causes each value to be evaluated and the result to be left on the evaluation stack and, if necessary, the value stack. This verb is used in conjunction with the RESTORE verb.

Incorrect entries are left on the evaluation stack if the SAVE and RESTORE verbs are performed in different procedures.

SDL Syntax:

Syntax Semantics:

value
: This field can be any valid SDL identifier or expression that returns a value and specifies the value to be evaluated. The result is left on the evaluation stack.

Example:

```
SAVE (A, B, C);
   •
   •
   •
RESTORE (C, B, A);
```

SAVE_STATE

The SAVE_STATE verb causes the state of the interpreter to be stored in the RS_M_MACHINE field of the program run structure nucleus and to then continue execution.

SDL Syntax:

```
—— SAVE_STATE; ——————————————————————————————————————————————————————————————————┤
```

Example:

```
SAVE_STATE;
```

SEARCH_DIRECTORY

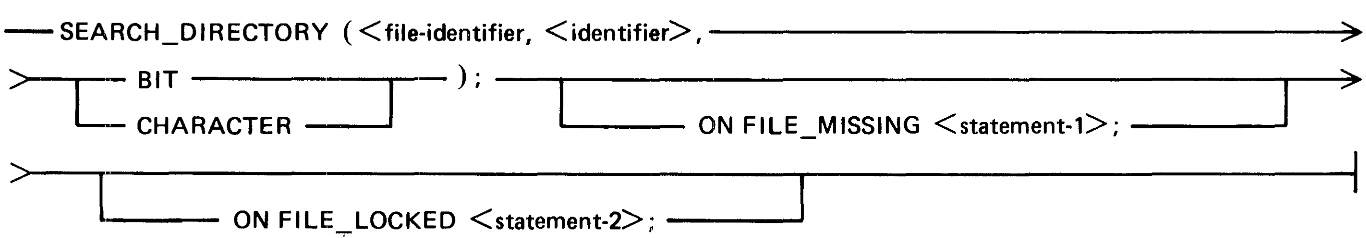
The SEARCH_DIRECTORY verb searches the disk directory for the requested file. If the file is found, information is gathered from the disk-file-header record and stored in <identifier>.

Table 9-5 shows the format of the information returned by the SEARCH_DIRECTORY verb. The format and content of the table are subject to change.

Table 9-5. Format of Information Returned from SEARCH_DIRECTORY

Item	BIT	CHARACTER
OPEN_TYPE	24	1
NO_USERS	24	2
RECORD_SIZE_IN_BITS	24	4
RECORDS_PER_BLOCK	24	4
EOF_POINTER	24	8
SEGMENTS_PER_AREA	24	8
USER_OPEN_OUTPUT	24	1
FILE_TYPE	24	2
PERMANENT_FLAG	24	2
BLOCKS_PER_AREA	24	6
AREAS_REQUESTED	24	3
AREA_COUNTER	24	3
SAVE_FACTOR	24	3
CREATION_DATE	24	5
LAST_ACCESS_DATE	24	5

SDL and UPL Syntax:



Syntax Semantics:

file-identifier

This field can be any valid SDL/UPL 30-character file identifier. The first 10 characters of <file-identifier> specify the pack identifier, the middle 10 characters specify the family identifier, and the last 10 characters specify the file identifier.

A one name file identifier must be left-justified in the middle 10 characters of <file-identifier>. All 30 characters must contain data, with each name left-justified and blank-filled on the right to the full 10-character length.

SEARCH_DIRECTORY

identifier
    This field can be any valid SDL/UPL identifier that has a BIT or CHARACTER data type. If the data type is BIT, the identifier must be 360 bits long. If the data type is CHARACTER, the identifier must be 59 bytes (characters) long.

BIT
    The keyword BIT causes the disk directory information to be stored in the identifier with a BIT data type.

CHARACTER
    The keyword CHARACTER causes the disk directory information to be stored in the identifier with a CHARACTER data type.

ON FILE_MISSING
    The keywords ON FILE MISSING cause <statement-1> to be performed if the file specified by <file-identifier> is not found in the disk directory.

ON FILE_LOCKED
    The keywords ON FILE_LOCKED cause <statement-2> to be performed if the file specified by <file-identifier> is locked.

statement-1
    This field can be any valid SDL/UPL statement and it is performed if the file specified by <file-identifier> is not found.

statement-2
    This field can be any valid SDL/UPL statement and it is performed if the file specified by <file-identifier> is locked.

SEARCH_DIRECTORY

Example Program:

```
DECLARE  01 DISK_FILE_HEADER        CHARACTER (59),
            03 OPEN_TYPE            CHARACTER (1),
            03 NO_USERS             CHARACTER (2),
            03 RECORD_SIZE          CHARACTER (4),
            03 RECORDS_PER_BLOCK    CHARACTER (4),
            03 EOF_POINTER          CHARACTER (8),
            03 SEGMENTS_PER_AREA    CHARACTER (8),
            03 USER_OPEN_OUTPUT     CHARACTER (1),
            03 FILE_TYPE            CHARACTER (2),
            03 PERMANENT_FLAG       CHARACTER (2),
            03 BLOCKS_PER_AREA      CHARACTER (6),
            03 AREAS_REQUESTED      CHARACTER (3),
            03 AREA_COUNTER         CHARACTER (3),
            03 SAVE_FACTOR          CHARACTER (3),
            03 CREATION_DATE        CHARACTER (5),
            03 LAST_ACCESS_DATE     CHARACTER (5),
         FILE_NAME                  CHARACTER (30);

FILE_NAME := "          SYSTEM    BACKUP    ";

SEARCH_DIRECTORY (FILE_NAME, DISK_FILE_HEADER, CHARACTER);
   ON FILE_MISSING DO;
                        DISPLAY ("SYSTEM/BACKUP NOT PRESENT");
                        DISPLAY ("GOOD BYE");
                        STOP;
                     END;
   ON FILE_LOCKED DO;
                       DISPLAY ("SYSTEM/BACKUP IS LOCKED");
                       DISPLAY ("GOOD BYE");
                       STOP;
                    END;

DISPLAY ("THE FOLLOWING IS THE DISK FILE HEADER FOR SYSTEM/BACKUP");
DISPLAY ("OPEN TYPE EQUALS " CAT OPEN_TYPE);
DISPLAY ("NUMBER OF USERS EQUALS " CAT NO_USERS);
DISPLAY ("RECORD SIZE EQUALS " CAT RECORD_SIZE CAT " BITS");
DISPLAY ("RECORDS PER BLOCK EQUALS " CAT RECORDS_PER_BLOCK);
DISPLAY ("END OF FILE EQUALS " CAT EOF_POINTER);
DISPLAY ("SEGMENTS PER AREA EQUALS " CAT SEGMENTS_PER_AREA);
DISPLAY ("USER OPEN OUTPUT EQUALS " CAT USER_OPEN_OUTPUT);
DISPLAY ("FILE TYPE EQUALS " CAT FILE_TYPE);
DISPLAY ("PERMANENT FLAG EQUALS " CAT PERMANENT_FLAG);
DISPLAY ("BLOCKS PER AREA EQUALS " CAT BLOCKS_PER_AREA);
DISPLAY ("NUMBER OF AREAS REQUESTED EQUALS " CAT AREAS_REQUESTED);
DISPLAY ("NUMBER OF AREAS EQUALS " CAT AREA_COUNTER);
DISPLAY ("SAVE FACTOR EQUALS " CAT SAVE_FACTOR);
DISPLAY ("CREATION DATE EQUALS " CAT CREATION_DATE);
DISPLAY ("LAST ACCESS DATE EQUALS " CAT LAST_ACCESS_DATE);
DISPLAY ("GOOD BYE");
STOP;
FINI;
```

SEARCH_DIRECTORY

Output from Example Program:

```
SEARCHO =1370 BOJ. PP=4, MP=4 TIME = 14:06:31.2
% SEARCHO =1370 THE FOLLOWING IS THE DISK FILE HEADER FOR SYSTEM/BACKUP
% SEARCHO =1370 OPEN TYPE EQUALS 0
% SEARCHO =1370 NUMBER OF USERS EQUALS 01
% SEARCHO =1370 RECORD SIZE EQUALS 1440
% SEARCHO =1370 RECORDS PER BLOCK EQUALS 0001
% SEARCHO =1370 END OF FILE EQUALS 00000092
% SEARCHO =1370 SEGMENTS PER AREA EQUALS 00000092
% SEARCHO =1370 USER OPEN OUTPUT EQUALS 0
% SEARCHO =1370 FILE TYPE EQUALS 08
% SEARCHO =1370 PERMANENT FLAG EQUALS 01
% SEARCHO =1370 BLOCKS PER AREA EQUALS 000092
% SEARCHO =1370 NUMBER OF AREAS REQUESTED EQUALS 001
% SEARCHO =1370 NUMBER OF AREAS EQUALS 001
% SEARCHO =1370 SAVE FACTOR EQUALS 000
% SEARCHO =1370 CREATION DATE EQUALS 79312
% SEARCHO =1370 LAST ACCESS DATE EQUALS 80136
% SEARCHO =1370 GOOD BYE
SEARCHO =1370 EOJ. TIME = 14:06:56.5
```

SEARCH_LINKED_LIST

The SEARCH_LINKED_LIST verb compares the value specified by <compare-value> with <compare-field>. If the comparison does not satisfy the relation, the next structure specified by <link-field> is used for the next comparison. This is an efficient way to search through a list of structures for a specific structure.

If the search succeeds, a 24-bit value is returned which is the base-relative address of the current structure. If the search fails, the value @FFFFFF@ is returned.

The last structure in the list must have all the bits equal to 1 for <link-field>.

SDL and UPL Syntax:

```
—— SEARCH_LINKED_LIST (<first-item>, <compare-field>, ——————————————>
>—— <compare-value>, <relation>, <link-field>); ————————————————————|
```

Syntax Semantics:

first-item
: This field can be any valid SDL/UPL identifier or expression that returns a value and specifies the first structure to be examined.

compare-field
: This field is a template which specifies the relative offset and size in the structure of the 24-bit field being compared with <compare-value>. A template is an identifier whose address is relative to the beginning of a structure rather than base relative. A field in a structure declared REMAPS BASE has such an address.

compare-value
: This field is the value that is compared with <compare-field>. <compare-value> is considered "on the left" in the compare relation.

relation
: This field specifies the desired relation in the comparison of <compare-field> and <compare-value>. The following is a list of the valid relation specifiers.

Relation	Description
<	less than
<=	less than or equal to
=	equal to
/=	not equal to
>=	greater than or equal to
>	greater than
LSS	less than
LEQ	less than or equal to
EQL	equal to
NEQ	not equal to
GEQ	greater than or equal to
GTR	greater than

SEARCH_LINKED_LIST

link-field

This field is a template which specifies the relative offset and size in the structure of the 24-bit (or less) field that contains the address of the next structure to be examined. <link-field> is examined if the comparison with the current structure failed. A template is an identifier whose address is relative to the beginning of a structure rather than base relative. A field in a structure declared REMAPS BASE has such an address.

Example:

```
BASE_RELATIVE_ADDR :=                   % Identifier BASE_RELATIVE_ADDR
 SEARCH_LINKED_LIST (FIRST_ADDRESS,     % is assigned the base-relative
   COMPARE_FIELD, COMPARE_VALUE,        % address of the structure that
   =, NEXT_LINK);                       % the search completed on and is
                                        % assigned the value @FFFFFF@ if
                                        % the search failed.
```

Example Program:

```
RECORD    TABLE
          DATA    FIXED,
          KEY     FIXED,
          LINK    BIT (24);

DECLARE ODT_INPUT         CHARACTER (4),
        COUNT             FIXED,
        RESULT            FIXED,
        COMPARE_VALUE     FIXED,
        T (1024)          TABLE;

COUNT := 0;
DO BUILD_LINKS FOREVER;
   IF COUNT = 1023 THEN UNDO BUILD_LINKS;
   T(COUNT).KEY := COUNT;
   T(COUNT).DATA := (TIME (COUNTER, BIT) MOD 1024);
   T(COUNT).LINK := DATA_ADDRESS (T(BUMP COUNT).DATA);
END BUILD_LINKS;
T(1023).LINK := @FFFFFF@;

DO FOREVER;
   DISPLAY ("ENTER ANY NUMBER FROM 0 TO 1023 OR ENTER BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE"
      THEN DO;
             DISPLAY ("GOOD BYE");
             STOP;
           END;
   COMPARE_VALUE := CONVERT (ODT_INPUT, FIXED);
   IF COMPARE_VALUE > 1023
      THEN DISPLAY (ODT_INPUT CAT " IS TOO LARGE");
      ELSE IF COMPARE_VALUE < 0
              THEN DISPLAY (ODT_INPUT CAT " IS TOO SMALL");
              ELSE DO;

                     RESULT := SEARCH_LINKED_LIST (T(0), KEY[0],
                               COMPARE_VALUE, =, LINK[0]);

                     IF RESULT = @FFFFFF@
                        THEN DISPLAY ("SEARCH FAILED");
                        ELSE DISPLAY ("RESULT EQUALS " CAT
                                      CONVERT (RESULT, CHARACTER));
                   END;
END;
FINI;
```

SEARCH_LINKED_LIST

```
% This example program shows one way to use the SEARCH_LINKED_LIST
% verb.  The program first builds a linked list using a table.
% The operator is then requested to enter any number between 0
% and 1023.  Using the accepted value, the program searches through
% the linked list for an equal condition and, if found, displays
% the base relative address of the beginning of the table entry that
% it found.  If the search fails, the program displays SEARCH FAILED.
% If BYE is entered, the program goes to end of job.
```

SEARCH_SDL_STACKS

The SEARCH_SDL_STACKS verb searches for a non-array or non-self-relative SDL descriptor whose address is within the given range of <compare-base> and <compare-top>. If the search is successful, @(1)1@ is returned. If the search is not successful, @(1)0@ is returned.

The SEARCH_SDL_STACKS verb is used by the SDL memory management intrinsics to determine which segments in memory can be rolled out to disk.

SDL Syntax:

```
—— SEARCH_SDL_STACKS (<stack-base>, <stack-top>, <compare-base>, ——————————>
>——— <compare-top>) ——————————————————————————————————————————————————————|
```

Syntax Semantics:

stack-base
: This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the beginning address of an SDL stack.

stack-top
: This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the address of the top of an SDL stack.

compare-base
: This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the address within the program at where the search is to begin.

compare-top
: This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the address within the program at where the search is to end.

Example:

```
DECLARE  LOWER          BIT (24),
         UPPER          BIT (24),
         RESULT         BIT (1);

LOWER := 0;
UPPER:= 10000;

RESULT := SEARCH_SDL_STACKS (CONTROL_STACK_TOP + CONTROL_STACK_BITS,
                             CONTROL_STACK_TOP, LOWER, UPPER);

IF NOT RESULT THEN DISPLAY ("SEARCH NOT SUCCESSFUL");
              ELSE DISPLAY ("SEARCH SUCCESSFUL");
STOP;
FINI;
```

SEARCH_SERIAL_LIST

The SEARCH_SERIAL_LIST verb searches a serial list of items beginning with the structure described by <first-item>. <compare-value> is compared with <compare-field> using the relation specified by <relation> until a match is found, or until <table-length> number of bits have been searched.

If <relation> is non-commutative, for example <, <=, >, >=, LSS, LEQ, GTR, or GEQ, the comparison is made as though <compare-value> is on the left of the relation.

If the search succeeds, the base-relative address of the item containing the successful <compare-field> is stored into <result-identifier> and the value @(1)1@ is returned. If the search fails, the end base-relative address of the table is stored into <result-identifier> and the value @(1)0 is returned.

SDL Syntax:

```
—— SEARCH_SERIAL_LIST (<compare-value>, <relation>, <compare-field> ——————————>

>———— , <first-item>, <table-length>, <result-identifier>) ; ——————————————————|
```

Syntax Semantics:

compare-value

This field is the value that is compared with <compare-field>. <compare-value> is considered the left portion of a compare relation.

relation

This field specifies the desired relation in the comparison of <compare-field> and <compare-value>. The following is a list of the valid relation specifiers.

Relation	**Description**
<	less than
<=	less than or equal to
=	equal to
/=	not equal to
>=	greater than or equal to
>	greater than
LSS	less than
LEQ	less than or equal to
EQL	equal to
NEQ	not equal to
GEQ	greater than or equal to
GTR	greater than

compare-field

This field is a template that gives the relative offset and size in the structure of the 24-bit field being compared with < compare-value >. A template is an identifier whose address is relative to the beginning of a structure rather than base relative. A field in a structure declared REMAPS BASE has such an address.

first-item

This field can be any valid SDL identifier or expression that returns a value and specifies the first structure to be examined.

SEARCH_SERIAL_LIST

table-length
This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the number of bits to search before stopping the search.

result-identifier
This field can be any valid SDL 24-bit identifier and contains the value of the end base-relative address of the table.

Example Program:

```
RECORD   TABLE
         DATA    FIXED,
         KEY     FIXED,
         LINK    BIT (24);

DECLARE ODT_INPUT       CHARACTER (4),
        COUNT           FIXED,
        RESULT          FIXED,
        COMPARE_VALUE   FIXED,
        T (1024)        TABLE;

COUNT := 0;
DO BUILD_LINKS FOREVER;
   IF COUNT = 1023 THEN UNDO BUILD_LINKS;
   T(COUNT).KEY := COUNT;
   T(COUNT).DATA := (TIME (COUNTER, BIT) MOD 1024);
   T(COUNT).LINK := DATA_ADDRESS (T(BUMP COUNT).DATA);
END BUILD_LINKS;
T(1023).LINK := @FFFFFF@;

DO FOREVER;
   DISPLAY ("ENTER ANY NUMBER FROM 0 TO 1023 OR ENTER BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE"
      THEN DO;
             DISPLAY ("GOOD BYE");
             STOP;
           END;
   COMPARE_VALUE := CONVERT (ODT_INPUT, FIXED);
   IF COMPARE_VALUE > 1023
      THEN DISPLAY (ODT_INPUT CAT " IS TOO LARGE");
      ELSE IF COMPARE_VALUE < 0
              THEN DISPLAY (ODT_INPUT CAT " IS TOO SMALL");

              ELSE IF SEARCH_SERIAL_LIST (COMPARE_VALUE, =, KEY[0],
                                          T(0), 73728, RESULT)

                      THEN DISPLAY ("RESULT EQUALS " CAT
                                        CONVERT (RESULT, CHARACTER));
                      ELSE DISPLAY ("SEARCH FAILED");
END;
FINI;
```

SEARCH_SERIAL_LIST

```
% This example program shows one way to use the SEARCH_SERIAL_LIST
% verb. The program first builds a serial linked list using a table.
% The operator is then requested to enter any number between 0
% and 1023. Using the accepted value, the program searches through
% the linked list for an equal condition and, if found, displays
% the base-relative address of the beginning of the table entry that
% it found. If the search fails, the program displays SEARCH FAILED.
% If BYE is entered, the program goes to end of job.
```

SEEK

The SEEK verb performs an actual hardware read and then stores the data in a buffer until the data is requested by a read operation. Use of the SEEK verb allows a programmer to overlap input/output (I/O) operations with processor operations.

When reading a file randomly and the next random record is known, the SEEK verb can be used to efficiently read random files. Specifying the SEEK verb immediately prior to a READ verb is less efficient than specifying the READ verb.

SDL and UPL Syntax:

```
—— SEEK <file-identifier> [<record-address-identifier>] ; ——————————————|
```

Syntax Semantics:

file-identifier
: This field can be any valid SDL/UPL file identifier and it specifies the file in which to perform the seek operation.

record-address-identifier
: This field can be any valid SDL/UPL identifier and specifies the record address within the file to seek. This identifier must be either a binary value of 24 bits or fewer in length, or an expression that generates a binary value.

Example:

```
SEEK DISKFILE [100];   % Causes a physical read of record number
                       % 100 from the disk file DISKFILE.  The data
                       % read is not made available until the program
                       % performs a READ statement.
```

SEEK

Example Program:

```
DECLARE  DATA                BIT (400),
         ODT_INPUT           CHARACTER (50),
         RECORD_ADDRESS      FIXED;

FILE DISKFILE (DEVICE = DISK RANDOM,
               RECORDS = 180/10,
               BUFFERS = 10);

RECORD_ADDRESS := 0;
OPEN DISKFILE INPUT;
SEEK DISKFILE [RECORD_ADDRESS];

DO FOREVER;

   DISPLAY ("ENTER BLANK TO DISPLAY THE NEXT RECORD OR BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                                DISPLAY ("GOOD BYE");
                                STOP;
                             END;
   READ DISKFILE [RECORD_ADDRESS] (DATA);
        ON EOF DO;
                 DISPLAY ("END OF FILE ENCOUNTERED -- GOOD BYE");
                 STOP;
              END;
        ON EXCEPTION DISPLAY ("RECORD " CAT RECORD_ADDRESS CAT
                              " NOT FOUND");
   BUMP RECORD_ADDRESS;

   SEEK DISKFILE [RECORD_ADDRESS];

   DISPLAY (CONVERT(DATA, CHARACTER));
END;

FINI;

% This example program uses the SEEK verb to physically read
% a record into the program's file buffer, and upon entering a blank
% message, the program performs a read operation to obtain
% the record.  Once the program performs a read operation, the
% program uses the SEEK verb to physically read the next record
% and displays the data within the record that was previously read.
```

SEGMENT_PAGE

The SEGMENT_PAGE verb divides the object code of a program into overlayable sections. When writing SDL/UPL programs, the programmer must explicitly segment programs if overlaying is to be allowed. If no SEGMENT_PAGE verbs appear, the entire program is compiled as one code segment. Run-time memory requirements for a program decrease when that program is segmented, because not all code segments must be resident in memory simultaneously.

When a program references a nonresident code segment, that code segment must be moved into main memory from disk. If no memory space is available, the newly called code segment is written (overlaid) into the space occupied by a less important code segment. The IMPORTANT keyword gives a code segment more protection from being overlaid.

The SEGMENT_PAGE verb can appear anywhere within an SDL/UPL program. The maximum number of code segments per page is 64. The maximum number of pages per program is 32.

There are two types of segmentation: permanent and temporary. Every SDL/UPL statement following a permanent segment statement is compiled to that code segment until another segment statement is encountered. Nonconsecutive groups of SDL/UPL statements can be compiled to the same code segment by specifying the same <segment-identifier> for each. The following example illustrates the use of the permanent segment statement.

```
SEGMENT_PAGE (XX);
DECLARE A1, A2, A3, A4;
PROCEDURE M;
    DECLARE B1, B2, B3;
    SEGMENT_PAGE (YY);
    PROCEDURE N;
       .
       .
       .
    END N;
    PROCEDURE P;
       .
       .
       .
    END P;
SEGMENT_PAGE (XX);
  .
  .
  .
END M;
  .
  .
  .
FINI;
```

SEGMENT_PAGE

Only procedures N and P have been compiled to the code segment labeled YY. The code segment labeled XX is segment zero and includes the remainder of the program. A SEGMENT_PAGE verb is temporary when it precedes any of the following verbs.

ACCESS_FILE_INFORMATION	RECEIVE
CASE	SEARCH_DIRECTORY
IF	SEND
OPEN	SPACE
READ	WRITE

The following example illustrates the use of temporary segmentation when an IF statement is specified.

```
SEGMENT_PAGE (A);
PROCEDURE X;
   .
   .
   .
   IF Y > 7
      THEN Y := 2;
      ELSE SEGMENT (B);
           DO SOME_FUNCTION;    %
                 .              % The DO-group
                 .              % SOME_FUNCTION is
                 .              % compiled to code
           END SOME_FUNCTION;   % segment B.
END X;
```

The DO-group SOME_FUNCTION in the preceding example is compiled into code segment B. Segment B automatically ends when the DO-group SOME_FUNCTION is terminated. All statements following the DO-group SOME_FUNCTION are compiled to segment A. Segment A is a permanent segment and segment B is a temporary segment.

DO-groups and procedures must begin and end in the same code segment. If this is not the case, the SDL/UPL compiler generates the following warning message and inserts code into the SDL/UPL program to bring the program back to the proper segment so that the DO-group and procedure can be exited correctly.

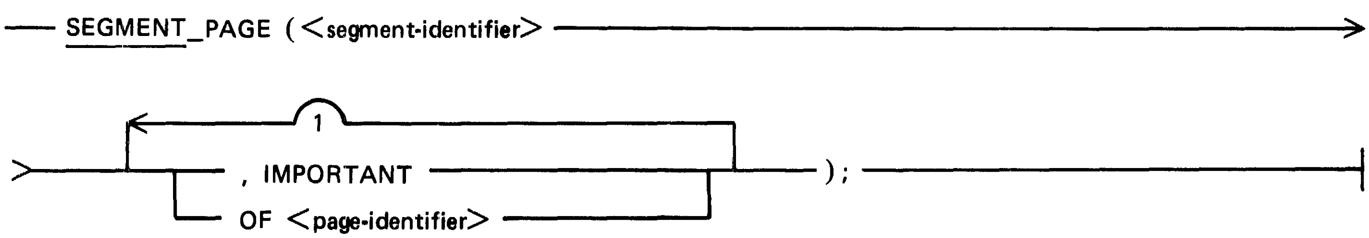
"DO GROUP" SHOULD TERMINATE IN SEGMENT IN WHICH IT BEGAN

PROCEDURE SHOULD TERMINATE IN SEGMENT IN WHICH IT BEGAN

Refer to the MCP MEMORY MANAGEMENT Appendix in the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982, for complete information on the Memory Management System.

SEGMENT_PAGE

SDL and UPL Syntax:



Syntax Semantics:

segment-identifier
This field can be any valid SDL/UPL identifier and specifies the name of a segment.

IMPORTANT
The keyword IMPORTANT causes the program code segment to remain in main memory. The segment is overlaid when the MCP requires additional memory space and no other portion of main memory is available for use.

OF
The keyword OF specifies that the <page-identifier> is to follow in the specification of the SEGMENT_PAGE verb.

page-identifier
This field can be any valid identifier and specifies the page in which the segment is to belong.

Example 1:

```
SEGMENT_PAGE (ZERO);                  % Assigns the DO-group A to the
  DO A;                               % code segment labeled ZERO.
     .
     .
     .
  END A;
```

Example 2:

```
SEGMENT_PAGE (TWO, IMPORTANT);        % Assigns the procedure PROC_X
  PROCEDURE PROC_X;                   % to the code segment identified
     .                                % as TWO.  This code segment is
     .                                % important.
     .
  END PROC_X;
```

Example 3:

```
SEGMENT_PAGE (TWO, IMPORTANT OF PAGE_1); % Assigns PROC_D to the
  PROC_D;                                % segment labeled TWO.
     .                                   % Also, this segment is an
     .                                   % important segment of the
     .                                   % page labeled PAGE_1.
  END D;
```

SEGMENT_PAGE

Example Program: For an example of the use of the SEGMENT__PAGE verb, refer to the LOCATION verb.

SKIP

The SKIP verb causes the line printer to skip to a specified channel number on the carriage control tape. These channel numbers correspond to holes punched in the carriage control tape. The channel numbers control the vertical spacing of records on a printed page and are defined by the carriage control tape on the printing device.

── SKIP <file-identifier> TO <channel-number> ; ──────────────────┤

Syntax Semantics:

file-identifier
: This field can be any valid SDL/UPL file identifier that is declared with a device type equal to PRINTER and specifies the file to perform the skip operation.

channel-number
: This field can be any valid SDL/UPL number between 1 and 12, inclusive and specifies the channel number to skip to on the carriage control tape.

Example 1:

```
SKIP LINE TO 1;      % The file labeled LINE must be an output file
                     % on the printing device.  The printing device
                     % advances to channel 1 (usually the top of a
                     % new page).
```

Example 2:

```
SKIP PRNT TO 12;     % The printing device advances to channel 12
                     % (usually at or near the end of a page).
```

SKIP

Example Program:

```
DECLARE  ODT_INPUT   CHARACTER (50);

FILE LINE  (DEVICE = PRINTER,
            RECORDS =132/1);

OPEN LINE OUTPUT;
DO FOREVER;
   DISPLAY ("ENTER CHARACTERS FOR THE PRINTER OR BYE TO GO TO EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                             DISPLAY ("GOOD BYE");
                             STOP;
                          END;
   SKIP LINE TO 1;
   WRITE LINE (ODT_INPUT);
         ON EXCEPTION DO;
                         DISPLAY ("EXCEPTION ON WRITE -- GOOD BYE");
                         STOP;
                     END;
END;

FINI;

% This example program accepts a record from the ODT and uses the
% SKIP verb to advance to channel 1 on the printing device prior
% to writing a record.  Enter BYE to send the program to end of job.
```

SORT

The following B 1000 utility programs can be invoked by the SORT verb.

SORT/MERGE
SORT/QSORT
SORT/TAPESORT
SORT/VSORT

These utility programs sort the specified input file and create the output file. <key-table> specifies the collating sequence desired in the sort. <sort-information-table> describes the options the sort is to use. Refer to B 1000 Systems SORT Reference Manual, form number 1090594 for a complete description of the B 1000 sort mechanism.

SDL and UPL Syntax:

```
── SORT ( <sort-information-table>, <key-table>, ──────────────────────>
>──── <input-file-identifier>, <output-file-identifier> ───────────────>
>──┬───────────────────────────────────────────┬── ); ─────────────────┤
   └───── , <translate-file-identifier> ───────┘
```

Syntax Semantics:

sort-information-table
: This field specifies the information required to sort a file. Refer to the B 1000 Systems SORT Reference Manual, form number 1090594, for the description and the format required in <sort-information-table>.

key-table
: This field specifies the sort key information required to sort a file. Refer to the B 1000 Systems SORT Reference Manual, form number 1090594, for the description and the format required for the <key-table>.

input-file-identifier
: This field can be any valid SDL/UPL file identifier that is declared in the file declaration section, and specifies the file in which to sort.

output-file-identifier
: This field can be any valid SDL/UPL file identifier that is declared in the file declaration section, and specifies the resulting file identifier of the sorted file.

translate-file-identifier
: This field can be any valid SDL/UPL file identifier and specifies the file to use for translating purposes.

Examples:

```
SORT (INFOR_TABLE, KEY_TABLE, IN_FILE, OUT_FILE);
SORT (INFOR_TABLE, KEY_TABLE, IN_FILE, OUT_FILE, TRANS_FILE);
```

SORT

Example Program:

```
DECLARE ODT_INPUT       CHARACTER (10),

        01 SORT_INFORMATION_TABLE,
            02  SORT_TYPE                BIT (2),
            02  SORT_HDWR                BIT (6),
            02  SORT_FILES               BIT (24),
            02  SORT_RECSIZE             BIT (24),
            02  SORT_IN_HDWR             BIT (6),
            02  SORT_IN_RECSIZE          BIT (24),
            02  SORT_IN_BLKSIZE          BIT (24),
            02  SORT_IN_CLOSE            BIT (12),
            02  SORT_IN_VARIABLE         BIT (1),
            02  SORT_OUT_HDWR            BIT (6),
            02  SORT_OUT_RECSIZE         BIT (24),
            02  SORT_OUT_BLKSIZE         BIT (24),
            02  SORT_OUT_CLOSE           BIT (12),
            02  SORT_OUT_VARIABLE        BIT (1),
            02  SORT_DELETING            BIT (1),
            02  SORT_STABILIZE           BIT (1),
            02  SORT_PARITY              BIT (1),
            02  SORT_RESTART             BIT (1),
            02  SORT_BIAS                BIT (7),
            02  SORT_RECORDS             BIT (24),
            02  SORT_TIMING              BIT (1),
            02  SORT_NUMBER_KEYS         BIT (5),
            02  SORT_TIME_IT             BIT (1),
            02  SORT_IN_OVERRIDE         BIT (1),
            02  FILLER                   BIT (6),
            02  SORT_KEY_LENGTH          BIT (16),
            02  FILLER                   BIT (16),
            02  SORT_PARTITION           BIT (24),
            02  SORT_DELETE_KEYS         BIT (4),
            02  SORT_DUPCHECK            BIT (1),
            02  SORT_TAGRPG              BIT (1),
            02  SORT_W1_PID              BIT (1),
            02  SORT_W2_PID              BIT (1),
            02  SORT_TAGCOBOL            BIT (1),
            02  FILLER                   BIT (15),
            02  SORT_MEMORY              BIT (24),
            02  SORT_TAGSEARCH           BIT (1),
            02  SORT_COLLATE             BIT (1),
            02  FILLER                   BIT (31),
            02  SORT_RESTART_JOB         BIT (24),
```

```
        01  SORT_KEY_TABLE                  BIT (1116),
            02  KEY (30)                    BIT (36),

        01  KEY_FIELD                       BIT (36),
            02  SIGN_FLAG                   BIT (1),
            02  DIRECTION                   BIT (1),
            02  FILLER                      BIT (1),
            02  COLLATE_KEY                 BIT (1),
            02  KEY_LENGTH                  BIT (12),
            02  KEY_DISPLACEMENT            BIT (20);

FILE IN    (DEVICE = DISK,
            RECORDS = 180/1),

     OUT   (DEVICE = DISK,
            RECORDS = 180/1);

SORT_TYPE := 0;  % USE SORT/VSORT
SORT_HDWR := @(1)0100001@;  % USE DISK FOR WORK FILES
SORT_FILES := 0;  % NO WORK TAPES
SORT_RECSIZE := @00005A0@;  % MAX RECORD SIZE = 180
SORT_IN_HDWR := @(1)0100001@;  % DISK
SORT_IN_RECSIZE := @00005A0@; % RECORD SIZE = 180
SORT_IN_BLKSIZE := @00000A@; % BLOCK SIZE = 10
SORT_IN_CLOSE := @400@;  % CLOSE WITH RELEASE
SORT_IN_VARIABLE := 0;  % NOT VARIABLE RECORDS
SORT_OUT_HDWR := @(1)0100001@;  % DISK
SORT_OUT_RECSIZE := @00005A0@;  % RECORDSIZE = 180
SORT_OUT_BLKSIZE := @00000A@;  % BLOCKSIZE = 10
SORT_OUT_CLOSE := @400@;  % CLOSE WITH RELEASE
SORT_OUT_VARIABLE := 0; % NOT VARIABLE RECORDS
SORT_DELETING := 0; % NO DELETING
SORT_STABILIZE := 0;  % SORT DUPLICATES IN ANY ORDER
SORT_PARITY := 0;  % DO NOT DISCARD RECORDS WITH PARITY ERROR
SORT_RESTART := 0;  % NO RESTART
SORT_BIAS := @(1)0110010@; % 50 PERCENT BIAS
SORT_RECORDS := @0003E8@; % 1000 RECORDS
SORT_TIMING := @(1)1@;  % REPORT SORT PARAMETERS
SORT_NUMBER_KEYS := @(1)00001@;  % 1 KEY
SORT_TIME_IT := @(1)1@;  % DISPLAY SORT TIME ON ODT
SORT_IN_OVERRIDE := 0;  % DO NOT USE INPUT BLOCKING
SORT_KEY_LENGTH := @50@;  % KEY LENGTH = 80 BITS OR 10 BYTES
SORT_PARTITION := 0;  % NO PARTITION
SORT_DELETE_KEYS := 0;  % NO INCLUDE OR DELETE KEYS
SORT_DUPCHECK := @(1)1@;  % REPORT DUPLICATE RECORDS
SORT_TAGRPG := 0;  % NOT RPG TAG FILE
SORT_W1_PID := 0;  % NO WORK PACK
SORT_W2_PID := 0;  % NO WORK PACK
SORT_TAGCOBOL := 0;  % NOT COBOL TAG FILE
SORT_MEMORY := @0493E0@;  % 300000 BITS OF MEMORY
SORT_TAGSEARCH := 0;  % NO TAG SEARCH
SORT_COLLATE := 0;  % NO COLLATE FILE
SORT_RESTART_JOB := 0; % NO RESTART
```

SORT

```
SIGN_FLAG := 0;   % NOT SIGNED
DIRECTION := 0;   % ASCENDING ORDER
COLLATE_KEY := 0;   % NO COLLATE TABLE
KEY_LENGTH := @50@;   % KEY LENGTH = 80 BITS OR 10 BYTES
KEY_DISPLACEMENT := @000000@;   % KEY STARTS IN FIRST POSITION

KEY (0) := KEY_FIELD;

DO FOREVER;
   DISPLAY ("ENTER INPUT FILE NAME OR ENTER BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                                  DISPLAY ("GOODBYE");
                                  STOP;
                              END;
   CHANGE IN TO (MULTI_FILE_ID := ODT_INPUT);
   DISPLAY ("ENTER OUTPUT FILE NAME OR ENTER BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                                  DISPLAY ("GOODBYE");
                                  STOP;
                              END;
   CHANGE OUT TO (MULTI_FILE_ID := ODT_INPUT);
   SORT (SORT_INFORMATION_TABLE, SORT_KEY_TABLE, IN, OUT);
END;
FINI;

% This example program shows the information required to
% use the SORT verb.  The program accepts from the ODT a
% 10-character file name for the input file and then accepts
% a second 10-character file name for the output file.  The
% input file must have a record size equal to 180 and blocking
% factor equal to 1.  If BYE is entered, the program goes to
% end of job.  Once the two file names are entered, the program
% invokes the SORT/VSORT sort utility program and sorts the
% file.
```

SORT__MERGE

The SORT__MERGE verb invokes the SORT/MERGE utility program. The SORT/MERGE program merges the specified input files and creates the output file. <key-table> specifies the collating sequence desired in the sort. <sort-information-table> describes the options the merge is to use. <merge-input-table> provides the relative file numbers of the files within the SDL/UPL program to merge. Refer to the B 1000 Systems SORT Reference Manual, form number 1090594, for a complete description of the B 1000 merge mechanism.

SDL and UPL Syntax:

```
── SORT_MERGE (<sort-information-table>, <key-table>, ──────────────────>

>──── <merge-input-table>, <output-file-identifier> ─────────────────────>

>──┬──────────────────────────────────────────┬── ); ───────────────────┤
   └────── , <translate-file-identifier> ──────┘
```

Syntax Semantics:

sort-information-table
This field specifies the information required to sort a file. Refer to the B 1000 Systems SORT Reference Manual, form number 1090594, for the description and the format required in <sort-information-table>.

key-table
This field specifies the sort key information required to sort a file. Refer to the B 1000 Systems SORT Reference Manual, form number 1090594, for the description and the format required for <key-table>.

merge-input-table
This field specifies the information required to sort a file. Refer to the B 1000 Systems SORT Reference Manual, form number 1090594, for the description and the format required for <merge-input-table>. <merge-input-table> specifies the relative file number within the SDL/UPL program to merge. A maximum of eight files can be merged.

output-file-identifier
This field can be any valid SDL/UPL file identifier that is declared in the file declaration section. It specifies the resulting file identifier of the sorted file.

translate-file-identifier
This field can be any valid SDL/UPL file identifier and specifies the file to use for translating purposes.

Example:

```
SORT_MERGE (INFOR_TABLE, KEY_TABLE, MERGE_INPUT_TABLE, OUT_FILE);
```

SORT_MERGE

Example Program:

```
DECLARE ODT_INPUT        CHARACTER (10),
        COUNTER          FIXED,

        01 SORT_INFORMATION_TABLE,
            02  SORT_TYPE                  BIT (2),
            02  SORT_HDWR                  BIT (6),
            02  SORT_FILES                 BIT (24),
            02  SORT_RECSIZE               BIT (24),
            02  SORT_IN_HDWR               BIT (6),
            02  SORT_IN_RECSIZE            BIT (24),
            02  SORT_IN_BLKSIZE            BIT (24),
            02  SORT_IN_CLOSE              BIT (12),
            02  SORT_IN_VARIABLE           BIT (1),
            02  SORT_OUT_HDWR              BIT (6),
            02  SORT_OUT_RECSIZE           BIT (24),
            02  SORT_OUT_BLKSIZE           BIT (24),
            02  SORT_OUT_CLOSE             BIT (12),
            02  SORT_OUT_VARIABLE          BIT (1),
            02  SORT_DELETING              BIT (1),
            02  SORT_STABILIZE             BIT (1),
            02  SORT_PARITY                BIT (1),
            02  SORT_RESTART               BIT (1),
            02  SORT_BIAS                  BIT (7),
            02  SORT_RECORDS               BIT (24),
            02  SORT_TIMING                BIT (1),
            02  SORT_NUMBER_KEYS           BIT (5),
            02  SORT_TIME_IT               BIT (1),
            02  SORT_IN_OVERRIDE           BIT (1),
            02  FILLER                     BIT (6),
            02  SORT_KEY_LENGTH            BIT (16),
            02  FILLER                     BIT (16),
            02  SORT_PARTITION             BIT (24),
            02  SORT_DELETE_KEYS           BIT (4),
            02  SORT_DUPCHECK              BIT (1),
            02  SORT_TAGRPG                BIT (1),
            02  SORT_W1_PID                BIT (1),
            02  SORT_W2_PID                BIT (1),
            02  SORT_TAGCOBOL              BIT (1),
            02  FILLER                     BIT (15),
            02  SORT_MEMORY                BIT (24),
            02  SORT_TAGSEARCH             BIT (1),
            02  SORT_COLLATE               BIT (1),
            02  FILLER                     BIT (31),
            02  SORT_RESTART_JOB           BIT (24),

        01  SORT_KEY_TABLE                 BIT (1116),
            02  KEY (30)                   BIT (36),
```

```
        01  KEY_FIELD                    BIT (36),
            02  SIGN_FLAG                BIT (1),
            02  DIRECTION                BIT (1),
            02  FILLER                   BIT (1),
            02  COLLATE_KEY              BIT (1),
            02  KEY_LENGTH               BIT (12),
            02  KEY_DISPLACEMENT         BIT (20),

         01 MERGE_INPUT_TABLE            BIT (80),
            02  FILLER                   BIT (8),
            02  MERGE_DISK_IN            BIT (8),
            02  MERGE_INPUT_FILE(8)      BIT (8);

FILE IN0  (DEVICE = DISK,
           RECORDS = 180/1),

     IN1  (DEVICE = DISK,
           RECORDS = 180/1),

     IN2  (DEVICE = DISK,
           RECORDS = 180/1),

     IN3  (DEVICE = DISK,
           RECORDS = 180/1),

     OUT  (DEVICE = DISK,
           RECORDS = 180/1);

SORT_TYPE := @(1)11@;  % USE SORT/MERGE
SORT_HDWR := 0;  % DOES NOT APPLY
SORT_FILES := @000004@;  % 4 INPUT FILES
SORT_RECSIZE := @0005A0@;  % MAX RECORD SIZE = 180
SORT_IN_HDWR := @(1)010001@;  % DISK
SORT_IN_RECSIZE := @0005A0@; % RECORD SIZE = 180
SORT_IN_BLKSIZE := @00000A@; % BLOCK SIZE = 10
SORT_IN_CLOSE := @400@;  % CLOSE WITH RELEASE
SORT_IN_VARIABLE := 0;  % NOT VARIABLE RECORDS
SORT_OUT_HDWR := @(1)010001@;  % DISK
SORT_OUT_RECSIZE := @0005A0@;  % RECORDSIZE = 180
SORT_OUT_BLKSIZE := @00000A@;  % BLOCKSIZE = 10
SORT_OUT_CLOSE := @400@;  % CLOSE WITH RELEASE
SORT_OUT_VARIABLE := 0; % NOT VARIABLE RECORDS
SORT_DELETING := 0; % NO DELETING
SORT_STABILIZE := 0;  % SORT DUPLICATES IN ANY ORDER
SORT_PARITY := 0;  % DO NOT DISCARD RECORDS WITH PARITY ERROR
SORT_RESTART := 0;  % NO RESTART
SORT_BIAS := @(1)0110010@; % 50 PERCENT BIAS
SORT_RECORDS := @0003E8@; % 1000 RECORDS
SORT_TIMING := @(1)1@;  % REPORT SORT PARAMETERS
SORT_NUMBER_KEYS := @(1)00001@;  % 1 KEY
SORT_TIME_IT := @(1)1@;  % DISPLAY SORT TIME ON ODT
SORT_IN_OVERRIDE := 0;  % DO NOT USE INPUT BLOCKING
SORT_KEY_LENGTH := @50@;  % KEY LENGTH = 180 BITS OR 10 BYTES
SORT_PARTITION := 0;  % NO PARTITION
```

SORT MERGE

```
SORT_DELETE_KEYS := 0;  % NO INCLUDE OR DELETE KEYS
SORT_DUPCHECK := @(1)1@;  % REPORT DUPLICATE RECORDS
SORT_TAGRPG := 0;  % NOT RPG TAG FILE
SORT_W1_PID := 0;  % NO WORK PACK
SORT_W2_PID := 0;  % NO WORK PACK
SORT_TAGCOBOL := 0;  % NOT COBOL TAG FILE
SORT_MEMORY := @493E0@; % 300000 BIT OF MEMORY
SORT_TAGSEARCH := 0;  % NO TAG SEARCH
SORT_COLLATE := 0;  % NO COLLATE FILE
SORT_RESTART_JOB := 0; % NO RESTART

SIGN_FLAG := 0;  % NOT SIGNED
DIRECTION := 0;  % ASCENDING ORDER
COLLATE_KEY := 0;  % NO COLLATE TABLE
KEY_LENGTH := @50@;  % KEY LENGTH = 180 BITS OR 10 BYTES
KEY_DISPLACEMENT := @000000@;  % KEY STARTS IN FIRST POSITION

KEY (0) := KEY_FIELD;

MERGE_DISK_IN := @(1)00000100@;  % 4 INPUT FILES ON DISK
MERGE_INPUT_FILE(0) := @(1)00000000@;  % RELATIVE FILE 0
MERGE_INPUT_FILE(1) := @(1)00000001@;  % RELATIVE FILE 1
MERGE_INPUT_FILE(2) := @(1)00000010@;  % RELATIVE FILE 2
MERGE_INPUT_FILE(3) := @(1)00000010@;  % RELATIVE FILE 3

COUNTER := 0;
DO FOREVER;
   DO ENTER_INPUT_FILENAME FOREVER;
      DISPLAY ("ENTER INPUT FILE NAME -- NUMBER " CAT
               DECIMAL (COUNTER, 1) CAT " OR ENTER BYE FOR EOJ");
      ACCEPT ODT_INPUT;
      IF ODT_INPUT = "BYE" THEN DO;
                                    DISPLAY ("GOODBYE");
                                    STOP;
                                 END;
      IF COUNTER = 0 THEN CHANGE IN0 TO (MULTI_FILE_ID := ODT_INPUT);
      IF COUNTER = 1 THEN CHANGE IN1 TO (MULTI_FILE_ID := ODT_INPUT);
      IF COUNTER = 2 THEN CHANGE IN2 TO (MULTI_FILE_ID := ODT_INPUT);
      IF COUNTER = 3 THEN CHANGE IN3 TO (MULTI_FILE_ID := ODT_INPUT);
      IF ((BUMP COUNTER) = 4) THEN UNDO ENTER_INPUT_FILENAME;
   END ENTER_INPUT_FILENAME;

   DISPLAY ("ENTER OUTPUT FILE NAME OR ENTER BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                                DISPLAY ("GOODBYE");
                                STOP;
                             END;
   CHANGE OUT TO (MULTI_FILE_ID := ODT_INPUT);
   SORT_MERGE (SORT_INFORMATION_TABLE, SORT_KEY_TABLE,
               MERGE_INPUT_TABLE, OUT);
END;
FINI;
```

```
% This example program uses the SORT_MERGE verb to merge four
% input files to create one output file.  The program accepts
% from the ODT the names of each input file and the name of the
```

SORT_SEARCH

The SORT_SEARCH verb is used only by the SORT programs and provides the information required to evaluate a record for sorting purposes. <first-table-entry-address> contains the address in an array of records of the first record to examine and <limit> specifies the last record to be examined.

SDL and UPL Syntax:

```
── SORT_SEARCH ( <first-table-entry-address>, <limit> ) ; ──────────────┤
```

Syntax Semantics:

first-table-entry-address

This field can be any valid SDL/UPL literal, identifier, or expression that returns a value and specifies the base-relative address of the first entry in the table of records to be examined and the condition under which records are to be selected.

limit

This field can be any valid SDL/UPL literal, identifier, or expression that returns a value and specifies the last record to be examined.

SORT__STEP__DOWN

The SORT__STEP__DOWN verb provides the information necessary to compare two records. <record-1> and <record-2> are the first and second records to be compared. <key-table-address> specifies the sort key used in the comparison.

This verb is for SORT program use only.

SDL Syntax:

```
—— SORT_STEP_DOWN (<record-1>, <record-2>, <key-table-address>); ——————————————|
```

Syntax Semantics:

record-1
: This field can be any valid SDL literal, identifier, or expression and specifies the first of two records that are to be compared.

record-2
: This field can be any valid SDL literal, identifier, or expression and specifies the second of two records that are to be compared.

key-table-address
: This field can be any valid SDL literal, identifier, or expression that returns a 24-bit value and specifies the address of the key table that the sort key uses for the comparison.

SORT__SWAP

The SORT__SWAP verb exchanges the values of two identifiers in memory without allocating a temporary storage area.

SDL and UPL Syntax:

```
── SORT_SWAP ( <identifier-1>, <identifier-2> ) ; ──────────────────────────┤
```

Syntax Semantics:

identifier-1
: This field can be any valid SDL/UPL identifier and specifies the first of two fields to be exchanged.

identifier-2
: This field can be any valid SDL/UPL identifier and specifies the second of two fields to be exchanged.

Example:

```
DECLARE   A   CHARACTER (10),
          B   CHARACTER (10);

A := "18";
B := "4982";
SORT_SWAP (A, B);    % Exchanges the values contained in identifiers
                     % A and B.
```

Figure 9-5 shows the contents of identifiers A and B before and after the SORT__SWAP operation.

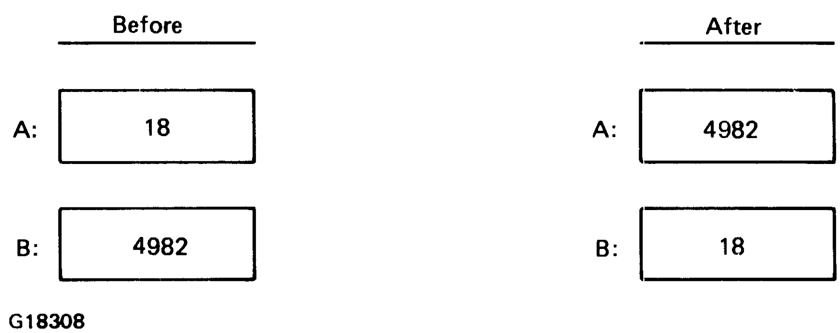


Figure 9-5. Contents of A and B Before/After SORT__SWAP Operation

SORT_SWAP

Example Program:

```
DECLARE    INPUT1  CHARACTER (10),
           INPUT2  CHARACTER (10);

DISPLAY ("ENTER THE FIRST 10 CHARACTERS");
ACCEPT INPUT1;
DISPLAY ("ENTER THE SECOND 10 CHARACTERS");
ACCEPT INPUT2;
DISPLAY ("VALUE OF INPUT1 BEFORE = " CAT INPUT1);
DISPLAY ("VALUE OF INPUT2 BEFORE = " CAT INPUT2);

SORT_SWAP (INPUT1, INPUT2);

DISPLAY ("VALUE OF INPUT1 AFTER  = " CAT INPUT1);
DISPLAY ("VALUE OF INPUT2 AFTER  = " CAT INPUT2);
DISPLAY ("GOOD BYE");
STOP;
FINI;

% This example program accepts two 10-character fields from
% the ODT, displays the values of the fields before performing
% the SORT_SWAP verb, and displays the values of the fields
% after performing the SORT_SWAP verb.
```

SORT_UNBLOCK

The SORT_UNBLOCK verb moves a record to and from a buffer and updates the buffer pointer and block count. This verb normally returns a 0 (zero). When the block count goes to 0 (zero), this verb restores the original buffer pointer and block count and returns @(1)1@. If the verb returns @(1)1@, the input/output (I/O) operation can take place.

A bit in the mini-FIB indicates to the SORT_UNBLOCK operation to create sort tags. If this bit is TRUE, the SORT_UNBLOCK operation uses the sort key table and selects only the key information to move from the buffer. A value in the mini-FIB represents the length of the receiving field.

This verb is for SORT program use only.

SDL Syntax:

```
—— SORT_UNBLOCK (<mini-FIB-address>, <length>, <source>, ——————————————>
>——— <destination>); ——————————————————————————————————————————————————|
```

Syntax Semantics:

mini-FIB-address
: This field can be any valid SDL identifier or expression that generates an address and specifies the address of the mini-FIB used by the SORT program.

length
: This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the length of the destination field.

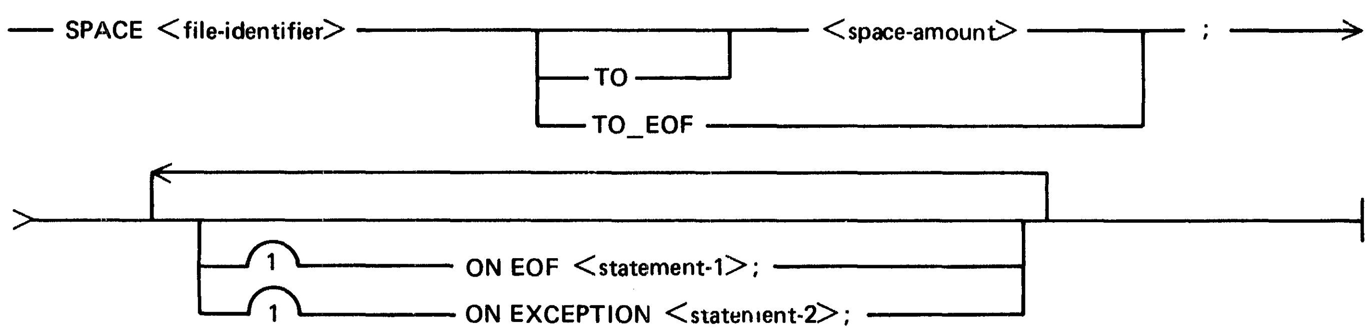
source
: This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the buffer from which the record is moved.

destination
: This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the buffer to which the record is moved.

SPACE

The SPACE verb causes the SDL/UPL program to position the file's current record pointer to the record specified by <space-amount> if the keyword TO is specified, or to skip the number of records specified by <space-amount> if the the keyword TO is not specified.

SDL and UPL Syntax:



Syntax Semantics:

file-identifier
: This field can be any valid SDL/UPL file identifier and specifies the file on which to perform the space operation.

TO
: The keyword TO specifies that skipping to the record number specified by integer, identifier, or expression is to be performed. The value of <space-amount> must be positive.

TO_EOF
: The keyword TO_EOF causes the SDL/UPL program to skip to the end-of-file record within the file.

space-amount
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value and specifies the number of records to skip over or the specific record to skip to in a sequential, fixed-length file. The value of <space-amount> must be positive.

ON EOF
: The keywords ON EOF cause the SDL/UPL program to perform <statement-1> if the SPACE operation results in reaching the end-of-file record.

ON EXCEPTION
: The keywords ON EXCEPTION cause the SDL/UPL program to perform <statement-1> if the SPACE operation cannot be completed because of an error condition.

statement-1
: This field can be any valid SDL/UPL statement and is performed when the program encounters the end-of-file record.

statement-2
: This field can be any valid SDL/UPL statement and is performed when the program encounters an exception in the file.

SPACE

Example 1:

```
SPACE LINE 3;                  % The LINE file is spaced three print
                               % lines on the line printer.
```

Example 2:

```
SPACE TAPEFILE TO X;           % The TAPEFILE file skips to the tape
      ON EOF STOP;             % record specified by the binary value
                               % of the identifier X.  If the end-of-file
                               % record is encountered, the program goes
                               % to end of job.
```

Example 3:

```
SPACE DISKFILE TO BUMP X;      % The DISKFILE file skips to the disk
      ON EOF STOP;             % record specified by the binary value of
      ON EXCEPTION STOP;       % BUMP X.  If the end-of-file record or a
                               % parity error occurs, the program goes to
                               % end of job.
```

Example Program:

```
DECLARE    ODT_INPUT      CHARACTER (10),
           DISK_RECORD    CHARACTER (180);

FILE IN (DEVICE = DISK,
         RECORDS = 180/1,
         USE_INPUT_BLOCKING);

DISPLAY ("ENTER 10-CHARACTER FILE NAME OR ENTER BYE FOR EOJ");
ACCEPT ODT_INPUT;
IF ODT_INPUT = "BYE" THEN DO;
                             DISPLAY ("GOOD BYE");
                             STOP;
                           END;
CHANGE IN TO (MULTI_FILE_ID := ODT_INPUT);
OPEN IN WITH INPUT;
     ON FILE_MISSING DO;
                        DISPLAY ("FILE " CAT ODT_INPUT CAT
                                 " NOT PRESENT -- GOOD BYE");
                        STOP;
                     END;
DO FOREVER;
   DISPLAY ("ENTER THE RECORD NUMBER TO SKIP TO OR ENTER BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                                CLOSE IN WITH RELEASE;
                                DISPLAY ("GOOD BYE");
                                STOP;
                             END;

   SPACE IN TO CONVERT (ODT_INPUT, FIXED);
    ON EOF DO;
             DISPLAY ("EOF ENCOUNTERED ON SPACE -- GOOD BYE");
             STOP;
           END;
    ON EXCEPTION DO;
                   DISPLAY ("PARITY ENCOUNTERED ON SPACE -- GOOD BYE");
                   STOP;
                 END;

   READ IN (DISK_RECORD);
     ON EXCEPTION DO;
                    DISPLAY ("PARITY ENCOUNTERED ON READ -- GOOD BYE");
                    STOP;
                  END;
   DISPLAY ("THE CONTENTS OF THE DISK RECORD ARE");
   DISPLAY (DISK_RECORD);
END;

FINI;
```

SPACE

```
% This example program uses the SPACE verb to position the
% disk file to the relative record number that is accepted
% from the ODT.  The program first accepts a 10-character
% file name from the ODT, then accepts the record number within
% the file to be displayed.  If BYE is entered, the program goes
% to end of job.  If the file requested is not present, or the
% program encounters a parity error while spacing, or the
% program encounters the end-of-file record, the program
% goes to end of job.
```

SPO__INPUT__PRESENT

The SPO__INPUT__PRESENT verb returns the value @(1)1@ if ODT input is present and returns the value @(1)0@if ODT input is not present.The SPO__INPUT__PRESENT verb assures that the ACCEPT verb has input, and does not suspend the program waiting for ODT input.

SDL and UPL Syntax:

```
—— SPO_INPUT_PRESENT ——————————————————————————————————————————|
```

Example:

```
DECLARE  BOOLEAN  BIT (1);        % The identifier BOOLEAN is assigned
BOOLEAN := SPO_INPUT_PRESENT;     % the value @(1)1@ if ODT input is
                                  % queued for the program and the
                                  % value @(1)0@ if ODT input is not
                                  % queued for the program.
```

Example Program:

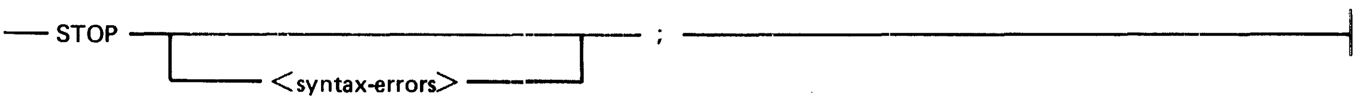
```
DECLARE ODT_INPUT CHARACTER (50);
DO FOREVER;
   IF SPO_INPUT_PRESENT
      THEN DO;
              ACCEPT ODT_INPUT;
              IF ODT_INPUT ="BYE" THEN STOP;
              DISPLAY (ODT_INPUT);
           END;
      ELSE IF NOT WAIT (TIME_TENTHS (100))
              THEN DISPLAY ("10 SECONDS HAVE EXPIRED");
END;
FINI;

% This example program uses the SPO_INPUT_PRESENT verb to check
% for any message in the ODT queue.  If there is a message, the
% program accepts the message and displays it on the ODT.
% If there is no message, the program waits 10 seconds for a
% message.  If no message is entered, the program displays
% 10 SECONDS HAVE EXPIRED on the ODT and continues to wait another
% 10 seconds.  If BYE is entered, the program goes to end of job.
```

STOP

The STOP verb causes the programmatic end of a program and notifies the MCP that the program has finished executing. The STOP and the FINI verbs have different functions. The FINI verb is the final statement in a SDL/UPL source program and marks the physical end of the source file.

<syntax-errors> is for use by B 1000 SDL/UPL compilers and causes the MCP to display the value as the number of syntax errors encountered when compiling a program. The value is displayed in the end-of-job message on the ODT.

SDL and UPL Syntax:



Syntax Semantics:

syntax-errors
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value and specifies the number of syntax errors that occurred.

Examples:

```
STOP;             % Causes the program to discontinue executing.

STOP 10;          % Causes the program to discontinue executing
                  % and to notify the MCP to show in the end-of-job
                  % message that 10 syntax errors occurred.
```

Example Program:

```
DECLARE   ODT_INPUT    CHARACTER (10);

DISPLAY ("ENTER THE NUMBER OF SYNTAX ERRORS DESIRED IN THE EOJ"
         CAT " MESSAGE");
ACCEPT ODT_INPUT;
DISPLAY ("GOOD BYE");
IF ODT_INPUT = ""
   THEN STOP;
   ELSE STOP CONVERT(ODT_INPUT, FIXED);
FINI;

% This example program accepts from the ODT the number of syntax
% errors that are desired to be included in the MCP end-of-job
% message.  If zeros or blanks are entered, no syntax errors
% are included.
```

SUBBIT

The SUBBIT verb provides the capability to address one or more bits within a bit string.

The SDL/UPL compiler does not verify that <start-position> and <length> are within bounds. Instead, a range check is performed at execution time on <start-position> and <length>, and an out-of-bounds value causes the program to terminate with an INVALID SUBSTRING program abort. In other words, <start-position> must reference a position in the bit string and <length> must not specify more bits than exist between <start-position> and the end of the string.

If the SUBBIT verb appears to the left of a assignment operator, the SUBBIT verb is treated as an address generator. Truncation, fill, and data alignment are performed by the operator with a BIT data type being the destination data type. In other words, if the source field is not declared with a BIT data type, the alignment is to the right and is controlled by the value of <start-position> and the number of bits specified by the value of <length>.

If <start-position> and <length> are declared with a BIT data type, each is evaluated as being a binary number. For example, if a literal "1" is specified, the EBCDIC value "1" is @F1@. This value converts to a binary value of 241, which results in specifying 241 as the <start-position> or specifying 241 as the length.

SDL and UPL Syntax:

```
── SUBBIT (<string-identifier>, <start-position> ──┬──────────────┬── ) ──┤
                                                   └─ , <length> ─┘
```

UPL Syntax:

```
── SUBBIT (<string-identifier>, <start-position> ──┬──────────────┬── ) ──┤
                                                   └─ , <length> ─┘
```

Syntax Semantics:

string-identifier
: This field can be any valid SDL/UPL identifier or expression that returns a value. If <string-identifier> is an expression, the data type returned is assumed to be equal to BIT. <string-identifier> specifies the name of the character string to be scanned.

start-position
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value and specifies the first element of the new string. <start-position> is a zero-relative offset to the beginning of <string-identifier>.

length
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value and specifies the number of elements that are to be included in the new string beginning with <start-position>. If <length> is not specified, all of the string beginning with <start-position> is included in the new string. Padding and truncation follow the standard SDL/UPL rules. If length has a value equal to zero, no string of bits is returned.

SUBBIT

Example 1:

```
DECLARE SBIT FIXED;                % Identifier A is assigned the value
SBIT := @(1)001000@;               % equal to @(1)0@.
A := SUBBIT (SBIT, 23, 1);
```

Example 2:

```
DECLARE SBIT FIXED;                % Identifier A is assigned the value
SBIT := @(1)001000@;               % equal to @(1)1@.
A := SUBBIT (SBIT, 21, 1);
```

Example 3:

```
DECLARE SBIT BIT (1),              % Identifier AX2 is assigned a resulting
        AX2  BIT (9);              % value equal to @(1)100110000@.
SBIT := @(1)1101111001@;
AX2 := @(1)100010100@;
SUBBIT (AX2, 3) :=
   SUBBIT (SBIT, 3, 2);
```

Example 4:

```
DECLARE OBJ_CODE BIT (16),         % The rightmost eight bits of identifier
        SOC_CODE FIXED;            % SOC_CODE are assigned to the rightmost
SUBBIT (OBJ_CODE, 8, 8) :=         % eight positions of OBJ_CODE.
   SOC_CODE;
```

Example 5:

```
DECLARE  X  BIT (8),               % Identifier X is assigned the
         C  BIT (8);               % resulting value equal to
X := @(1)11111111@;                % @(1)00001111@.
C := @(1)00000000@;
SUBBIT (X, 4) :=
   SUBBIT (C, 0, 4);
```

SUBBIT

Example Program:

```
DECLARE     ODT_INPUT         CHARACTER (5),
            STRING            BIT (40),
            LENGTH            FIXED,
            START_POSITION    FIXED,
            DISPLAY_FIELD     CHARACTER (5);

DISPLAY ("ENTER ANY 5-CHARACTER STRING OR ENTER BYE FOR EOJ");
ACCEPT ODT_INPUT;
IF ODT_INPUT = "BYE" THEN DO;
                                DISPLAY ("GOOD BYE");
                                STOP;
                             END;
STRING := ODT_INPUT;
DO FOREVER;
   DO FOREVER;
      DISPLAY ("ENTER ANY OF THE FOLLOWING 2-CHARACTER NUMBERS FOR"
                 CAT " THE START POSITION OR ENTER BYE FOR EOJ -- 0,"
                 CAT " 8, 16, 24, 32");
      ACCEPT ODT_INPUT;
      IF ODT_INPUT = "BYE" THEN DO;
                                      DISPLAY ("GOOD BYE");
                                      STOP;
                                   END;
      START_POSITION := CONVERT(ODT_INPUT, FIXED);
      IF NOT (START_POSITION > 39) AND NOT (START_POSITION < 0)
      THEN UNDO;
      ELSE DISPLAY ("THE VALUE FOR START POSITION IS OUT OF RANGE");
   END;
   DO FOREVER;
      DISPLAY ("ENTER ANY OF THE FOLLOWING 2-CHARACTER NUMBERS FOR"
                CAT " THE LENGTH OR ENTER BYE FOR EOJ -- 0, 8, 16, 24"
                CAT " 32, 40");
      ACCEPT ODT_INPUT;
      IF ODT_INPUT = "BYE" THEN DO;
                                      DISPLAY ("GOOD BYE");
                                      STOP;
                                   END;
      LENGTH := CONVERT(ODT_INPUT, FIXED);
      IF NOT ((START_POSITION + LENGTH) > 40)
      THEN UNDO;
      ELSE DO;
              DISPLAY ("THE VALUE ENTERED FOR LENGTH IS OUT OF RANGE");
              DISPLAY ("LENGTH MUST NOT BE GREATER THAN "
                       CAT CONVERT((40 - START_POSITION), CHARACTER));
           END;
   END;

DISPLAY_FIELD := SUBBIT (STRING, START_POSITION, LENGTH);

DISPLAY ("THE SUBBIT VALUE IS " CAT DISPLAY_FIELD);
END;
```

SUBBIT

```
FINI;

% This example program uses the SUBBIT verb to display a partial
% character string in bits.  The program accepts from the ODT
% the character string, and then accepts two, 2-character numbers
% for the starting position and length.  The resulting partial
% character string is then displayed on the ODT.  If BYE is
% entered, the program goes to end of job.
```

SUBSTR

The SUBSTR verb provides the capability to address one or more characters within a character string.

The SDL/UPL compiler does not verify that <start-position> and <length> are within bounds. Instead, a range check is performed at execution time on <start-position> and <length>, and an out-of-bounds value causes the program to terminate with an INVALID SUBSTRING program abort. In other words, <start-position> must reference a position in the character string and <length> must not specify more characters than exist between <start-position> and the end of the string.

If the SUBSTR verb appears to the left of an assignment operator, it is treated as an address generator. Truncation, fill, and data alignment are performed by the SUBSTR verb and the destination data type is CHARACTER. In other words, if the source field is not declared with a CHARACTER data type, the alignment is to the right and is controlled by <start-position> and the number of characters specified by <length>. If the source field is declared with a CHARACTER data type, the alignment is left-justified to the position as specified by <start-position> and is controlled by the value of <start-position> and the number of characters in the value of <length>.

If <start-position> and <length> are declared with a CHARACTER data type, each is evaluated as a binary number. For example, if a literal one ("1") is specified, the EBCDIC value "1" is @F1@. This value converts to a binary value of 241, which results in specifying 241 as <start-position> or specifying 241 as the length.

A value of zero for <length> is valid and describes a null substring. Any attempt to assign data to a null string causes no data to be stored and no errors to be generated.

SDL and UPL Syntax:

```
—— SUBSTR (<string-identifier>, <start-position> ——————————┬————————————————┬—— ) ——|
                                                           └—— ,<length> ———┘
```

Syntax Semantics:

string-identifier
: This field can be any valid SDL/UPL identifier or expression that returns a value. If <string-identifier> is an expression, the data type of <string-identifier> is assumed to be equal to CHARACTER. <string-identifier> specifies the name of the character string to be scanned. If <string-identifier> is the name of a file, then a 24-bit integer value is generated, representing the file number of the file as it is declared in the source file.

start-position
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value and specifies the first element of the new string. <start-position> is a zero-relative offset to the beginning of <string-identifier>.

length
: This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value and specifies the number of elements that are to be included in the new string beginning with <start-position>. If <length> is not specified, all of the string beginning with <start-position> is included in the new string. Padding and truncation follow the standard SDL/UPL rules. If <length> has a value equal to 0 (zero), then no string of characters is returned. If <length> is omitted, <start-position> must be modulo 8.

SUBSTR

Example 1:

```
DECLARE ALFA  CHARACTER (26);
ALFA := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
X := SUBSTR(ALFA, 0, 1);          % Identifier X contains the value
                                  % equal to "A".
```

Example 2:

```
DECLARE ALFA  CHARACTER (26);
ALFA := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
X := SUBSTR(ALFA, 24);            % Identifier X contains the value
                                  % equal to "YZ".
```

Example 3:

```
N := 0;                           % Identifier N has a data type
DO ODC FOREVER;                   % equal to FIXED.  Identifier PRINT
   SUBSTR(PRINT, N, 1) :=         % contains every other letter in the
      SUBSTR(ALFA, 2 * N, 1);     % string, for example, A C E ... W Y.
   IF (2 * (BUMP N)) GTR 25
      THEN UNDO ODC;
END ODC;
```

Example 4:

```
ABC := "OPPOSITE";                % The value of identifier ABC is
CH := "VAULT";                    % changed from "OPPOSITE" to
SUBSTR(ABC, 0, 1) :=              % "APPOSITE".
    SUBSTR(CH, 1, 1);
```

Example 5:

```
X := "CHARACTER";                  % The value of identifier X
C := "COALITION";                  % becomes "CHARCOAL".
SUBSTR(X,4) := SUBSTR(C,0,4);
```

Example Program:

```
DECLARE    ODT_INPUT        CHARACTER (40),
           STRING           CHARACTER (40),
           LENGTH           FIXED,
           START_POSITION   FIXED,
           DISPLAY_FIELD    CHARACTER (40);

DISPLAY ("ENTER ANY 40-CHARACTER STRING OR ENTER BYE FOR EOJ");
ACCEPT ODT_INPUT;
IF ODT_INPUT = "BYE" THEN DO;
                             DISPLAY ("GOOD BYE");
                             STOP;
                          END;
```

```
STRING := ODT_INPUT;
DO FOREVER;
   DO FOREVER;
      DISPLAY ("ENTER ANY 2-CHARACTER NUMBER FOR THE START POSITION OR"
                 CAT " ENTER BYE FOR EOJ");
      ACCEPT ODT_INPUT;
      IF ODT_INPUT = "BYE" THEN DO;
                                 DISPLAY ("GOOD BYE");
                                 STOP;
                              END;
      START_POSITION := CONVERT(ODT_INPUT, FIXED);
      IF NOT (START_POSITION > 39) AND NOT (START_POSITION < 0)
      THEN UNDO;
      ELSE DISPLAY ("THE VALUE FOR START POSITION IS OUT OF RANGE");
   END;
   DO FOREVER;
      DISPLAY ("ENTER ANY 2-CHARACTER NUMBER FOR THE LENGTH OR ENTER"
                CAT " BYE FOR EOJ");
      ACCEPT ODT_INPUT;
      IF ODT_INPUT = "BYE" THEN DO;
                                 DISPLAY ("GOOD BYE");
                                 STOP;
                              END;
      LENGTH := CONVERT(ODT_INPUT, FIXED);
      IF NOT ((START_POSITION + LENGTH) > 40)
      THEN UNDO;
      ELSE DO;
              DISPLAY ("THE VALUE ENTERED FOR LENGTH IS OUT OF RANGE");
              DISPLAY ("LENGTH MUST NOT BE GREATER THAN "
                       CAT CONVERT((40 - START_POSITION), CHARACTER));
          END;
   END;

DISPLAY_FIELD := SUBSTR(STRING, START_POSITION, LENGTH);

DISPLAY ("THE SUBSTRING VALUE IS " CAT DISPLAY_FIELD);
END;
FINI;


% This example program uses the SUBSTR verb to display a
% substring of a character string.  The program accepts from
% the ODT the character string, and then accepts two, 2-character
% numbers for the starting position and length and displays on the
% ODT the substring that results.  If BYE is entered, the program
% goes to end of job.
```

SWAP

The SWAP verb returns the current value of <destination> and stores the value of <source> into <destination>. The value of <source> remains unchanged after the SWAP operation.

The length of <destination> determines the number of bytes of <source> that are stored into <destination>. If the length of <destination> is greater than 24 bits, then only the rightmost 24 bits of <source> are stored. If the length of <source> is less than <destination> and <destination> is less than or equal to 24 bits, <destination> is padded with leading zeros.

SDL and UPL Syntax:

```
── SWAP ( <destination>, <source> ──────────────────────────────────┤
```

SDL Syntax Semantics:

destination
: This field can be any valid SDL/UPL identifier and specifies the destination field of the SWAP operation.

source
: This field can be any valid SDL/UPL literal, identifier, or expression that returns a value and specifies the source field for the SWAP operation.

UPL Syntax Semantics:

Refer to the SORT_SWAP verb for the semantics of the UPL syntax.

Example 1:

```
DECLARE  A  FIXED,          % The value of identifier B is stored
         B  FIXED,          % into identifier A, and identifier C
         C  FIXED;          % is assigned the value of identifier A.
A := 99;
B := 1;
C := SWAP (A, B);
```

Example 2:

```
DECLARE A  FIXED;           % The ELSE part of the statement is
A := 0;                     % evaluated, since the value of identifier
IF SWAP (A, 1)              % A was originally assigned a value of
THEN DO;                    % 0 (that is, FASLE).  At the end of the
       .                    % SWAP operation, the value 1 is stored
       .                    % into identifier A and the value 0 is
       .                    % returned to the top of the evaluation
     END;                   % stack.
ELSE DO;
       .
       .
       .
     END;
```

SWAP

Example Program:

```
DECLARE    ODT_INPUT     CHARACTER (3),
           ODT_SAVE      CHARACTER (3),
           SWAP_FIELD    CHARACTER (3);

DO FOREVER;
   DISPLAY ("ENTER 3 CHARACTERS FOR NEW VALUE OF ODT_INPUT OR ENTER"
            CAT " BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                                DISPLAY ("GOOD BYE");
                                STOP;
                             END;

   SWAP_FIELD := SWAP (ODT_SAVE, ODT_INPUT);

   DISPLAY ("THE VALUE OF ODT_INPUT = " CAT ODT_INPUT);
   DISPLAY ("THE VALUE OF ODT_SAVE = " CAT ODT_SAVE);
   DISPLAY ("THE VALUE OF SWAP_FIELD = " CAT SWAP_FIELD);
END;
FINI;

% This example program uses the SWAP verb to store the value
% accepted from the ODT in identifier ODT_SAVE and assigns the
% old value of ODT_SAVE to identifier SWAP_FIELD.  The value of
% identifiers ODT_INPUT, ODT_SAVE, and SWAP_FIELD are displayed
% on the ODT.  If BYE is entered, the program goes to end of job.
```

S_MEM_SIZE

The S_MEM_SIZE verb returns a 24-bit value which is the S-memory size in bits of the B 1000 computer system.

SDL Syntax:

```
—— S_MEM_SIZE ——————————————————————————————————————————————————|
```

Example:

```
DECLARE  MEMORY  BIT (24);   % Identifier MEMORY is assigned the
MEMORY := S_MEM_SIZE;        % value of the memory size of the
                             % B 1C00 computer system.
```

Example Program:

```
DISPLAY ("THE S-MEMORY SIZE EQUALS @" CAT
          CONVERT ((S_MEM_SIZE / 8), CHARACTER) CAT "@ BYTES");
STOP;
FINI;
```

Output from Example Program:

```
S_MEM_SIZO =6234 BOJ. PP=4, MP=4 TIME = 10:37:11.4
% S_MEM_SIZO =6234 THE S-MEMORY SIZE EQUALS @100000@ BYTES
S_MEM_SIZO =6234 EOJ. TIME = 10:37:16.7
```

THAW_PROGRAM

The THAW_PROGRAM verb resets the memory and rollout lock bits set by the FREEZE_-PROGRAM verb. The THAW_PROGRAM verb allows the run structure nucleus of the program to be moved in and out of memory as required by the MCP.

The THAW_PROGRAM verb has no effect if the memory and rollout lock bits are not set.

SDL and UPL Syntax:

```
—— THAW_PROGRAM; ——————————————————————————————————————————|
```

Example:

```
THAW_PROGRAM;        % Causes the run structure nucleus of the program
                     % to be moved in and out of memory as required by
                     % the MCP.
```

THREAD_VECTOR

The THREAD_VECTOR verb is used only by the SORT program.

SDL Syntax:

── THREAD_VECTOR (<table-address>, <index>); ──────────────┤

Syntax Semantics:

table-address
: This field can be any valid SDL literal, identifier, or expression that returns the table address of the table containing the information described in the INITIALIZE_VECTOR verb.

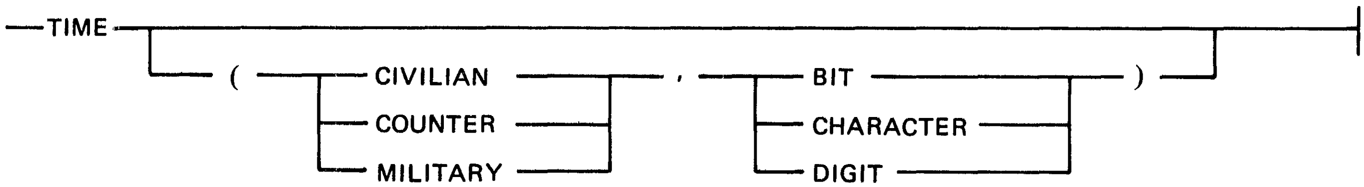
index
: This field can be any valid SDL literal, identifier, or expression that returns a value and specifies the offset from the beginning of the table to the next record to be used for comparison.

TIME

The TIME verb returns a bit or character string whose value is the current system time.

TIME and TIME(CIVILIAN, CHARACTER) are equivalent.

SDL and UPL Syntax:



Syntax Semantics:

CIVILIAN
: The keyword CIVILIAN causes the time to be returned in the HHMMSSTAP format, where HH is the hours, MM is the minutes, SS is the seconds, T is tenths of a second, and AP is AM or PM.

COUNTER
: The keyword COUNTER causes the time to be returned in the TTTTT format, where TTTTT is the time in tenths of seconds.

MILITARY
: The keyword MILITARY causes the time to be returned in the HHMMSST format, where HH is the hours, MM is the minutes, SS is the seconds, and T is tenths of a second.

BIT
: The keyword BIT specifies the time to be in the bit format. The following is the bit format for CIVILIAN, COUNTER, and MILITARY time.

```
01 CIVILIAN     BIT (36),
   03 HH        BIT (4),
   03 MM        BIT (6),
   03 SS        BIT (6),
   03 T         BIT (4),
   03 AP        BIT (16);

01 COUNTER      BIT (20);

01 MILITARY     BIT (21),
   03 HH        BIT (5),
   03 MM        BIT (6),
   03 SS        BIT (6),
   03 T         BIT (4);
```

TIME

CHARACTER

The keyword CHARACTER specifies the time to be in the character format. The following is the character format for CIVILIAN, COUNTER, and MILITARY time.

```
01 CIVILIAN     CHARACTER (9),
   03  HH       CHARACTER (2),
   03  MM       CHARACTER (2),
   03  SS       CHARACTER (2),
   03  T        CHARACTER (1),
   03  AP       CHARACTER (2);

01 COUNTER      CHARACTER (6);

01 MILITARY     CHARACTER (7),
   03  HH       CHARACTER (2),
   03  MM       CHARACTER (2),
   03  SS       CHARACTER (2),
   03  T        CHARACTER (1);
```

DIGIT

The keyword DIGIT specifies the time to be in the digit format. The following is the digit format for CIVILIAN, COUNTER, and MILITARY time.

```
01 CIVILIAN     BIT (44),
   03  HH       BIT (8),
   03  MM       BIT (8),
   03  SS       BIT (8),
   03  T        BIT (4),
   03  AP       BIT (16);

01 COUNTER      BIT (24);

01 MILITARY     BIT (28),
   03  HH       BIT (8),
   03  MM       BIT (8),
   03  SS       BIT (8);
```

Example:

```
DECLARE  CIVILIAN_TIME  CHARACTER (9),
         COUNTER_TIME   BIT (20),
         MILITARY_TIME  BIT (28);

CIVILIAN_TIME := TIME(CIVILIAN, CHARACTER);
COUNTER_TIME  := TIME(COUNTER, DIGIT);
MILITARY_TIME := TIME(MILITARY, BIT);
```

TIME

If the current system time is 11:30:50.4 AM, then CIVILIAN_TIME, COUNTER_TIME, and MILITARY_TIME have the following bit and hexadecimal values.

```
CIVILIAN_TIME = @(1)1111 0001 1111 0001 1111 0011 1111 0000 1111
                        0101 1111 0000 1111 0100 1100 0001 1101 0100@
              = @(4)F1F1F3F0F5F0F4C1D4@

COUNTER_TIME = @(1)0110 0101 0011 0010 1010@
             = @(4)6532A@

MILITARY_TIME = @(1)0001 0001 0011 0000 0101 0000 0100@
              = @(4)1130504@
```

Example Program:

```
DECLARE 01 CIVILIAN_TIME  CHARACTER (9),
           03 CIV_HH      CHARACTER (2),
           03 CIV_MM      CHARACTER (2),
           03 CIV_SS      CHARACTER (2),
           03 CIV_T       CHARACTER (1),
           03 CIV_AP      CHARACTER (2),
        01 COUNTER_TIME   CHARACTER (6),
        01 MILITARY_TIME  CHARACTER (7),
           03 MIL_HH      CHARACTER (2),
           03 MIL_MM      CHARACTER (2),
           03 MIL_SS      CHARACTER (2),
           03 MIL_T       CHARACTER (1);

CIVILIAN_TIME := TIME (CIVILIAN, CHARACTER);
COUNTER_TIME  := TIME (COUNTER, CHARACTER);
MILITARY_TIME := TIME (MILITARY, CHARACTER);

IF CIV_AP = "AM" THEN
DISPLAY ("THE CURRENT SYSTEM TIME IN CIVILIAN FORMAT IS " CAT CIV_HH
         CAT " HOURS, " CAT CIV_MM CAT " MINUTES, " CAT CIV_SS CAT
         " SECONDS, AND " CAT CIV_T CAT " TENTHS OF A SECOND IN"
         CAT " THE MORNING");
IF CIV_AP = "PM" THEN
DISPLAY ("THE CURRENT SYSTEM TIME IN CIVILIAN FORMAT IS " CAT CIV_HH
         CAT " HOURS, " CAT CIV_MM CAT " MINUTES, " CAT CIV_SS CAT
         " SECONDS, AND " CAT CIV_T CAT " TENTHS OF A SECOND IN"
         CAT " THE AFTERNOON");
DISPLAY ("THE CURRENT SYSTEM TIME IN COUNTER FORMAT IS " CAT
         COUNTER_TIME CAT " TENTHS OF A SECOND");
DISPLAY ("THE CURRENT SYSTEM TIME IN MILITARY FORMAT IS " CAT
         MIL_HH CAT " HOURS, " CAT MIL_MM CAT " MINUTES, " CAT
         MIL_SS CAT " SECONDS, AND " CAT MIL_T CAT
         " TENTHS OF A SECOND");
STOP;
FINI;

% This example program uses the TIME verb with the civilian,
% counter, and military format and displays the current system
% using each format with a data type equal to CHARACTER.
```

TIME

Output from Example Program:

```
TIMEO =4186 BOJ. PP=4, MP=4 TIME = 12:27:39.9
% TIMEO =4186 THE CURRENT SYSTEM TIME IN CIVILIAN FORMAT IS 12
    HOURS, 27 MINUTES, 41 SECONDS, AND 6 TENTHS OF A SECOND IN
   THE AFTERNOON
% TIMEO =4186 THE CURRENT SYSTEM TIME IN COUNTER FORMAT IS 448
   616 TENTHS OF A SECOND
% TIMEO =4186 THE CURRENT SYSTEM TIME IN MILITARY FORMAT IS 12
    HOURS, 27 MINUTES, 41 SECONDS, AND 6 TENTHS OF A SECOND
TIMEO =4186 EOJ. TIME = 12:27:47.5
```

TIMER

The TIMER verb returns a 24-bit value that is the current setting of the time register.

SDL Syntax:

```
—— TIMER ——————————————————————————————————————————————————————————————————|
```

Example:

```
DECLARE X    BIT (24);       % Identifier X is assigned the current
X := TIMER;                  % value of the time register.
```

Example Program:

```
DISPLAY ("THE VALUE OF THE TIME REGISTER IS " CAT
          CONVERT (TIMER, CHARACTER));
STOP;
FINI;

% This example program displays the current setting of the
% time register.
```

Output from Example Program:

```
TIMERO =2270 BOJ. PP=4, MP=4 TIME = 08:40:15.0
% TIMERO =2270 THE VALUE OF THE TIME REGISTER IS 04F4F5
TIMERO =2270 EOJ. TIME = 08:40:18.0
```

TRACE

The TRACE verb causes the SDL instructions of the normal state program to be traced on the line printer. Specifying the NOTRACE verb turns off the trace. The tracing is effective only when the program is executed with the SDL trace interpreter.

The following is the meaning of each of the 10 bits in <trace-option>.

Bit	Use
0	Trace all commands except those which modify data or change the program pointer stack. This bit applies to normal state programs.
1	Trace all commands which modify data items, for example, CLR, SNDL, and so forth. This bit applies to normal state programs.
2	Trace all commands which change the program pointer stack; for example, IFTH, CASE, EXIT, and so forth. This bit applies to normal state programs.
3	Not used.
4-6	These bits have the same respective meanings as bits 0 through 2 and are used only for the MCP. Several MCP routines (for example, GETSPACE, FORGETSPACE, and so forth) are not traced.
7-9	These bits have the same respective meanings as bits 0 through 2 and are used only for the MCP. The MCP routines not traced by setting bits 4 through 6 are traced.

SDL Syntax:

```
-----------+--- TRACE ---+-------------------------+--- ; ----------------|
           |             |                         |
---- NO ---+             +--- ( <trace-options> ) -+
```

Syntax Semantics:

trace-options
This field can be any valid SDL literal, identifier, or expression that returns a value and specifies which trace option to use. The leftmost 10 bits specify which option to use.

Examples:

```
NOTRACE;          % Turns off the tracing of the program.

TRACE;            % Turns on the tracing of the program.

TRACE (2);        % Turns on the tracing of the program and
                  % and traces commands which change the
                  % program pointer stack.
```

TRANSLATE

The TRANSLATE verb translates each item in <source-identifier>, using the <translate-table>, and stores the value in <result-identifier>. The translation continues until one of the following conditions occurs.

1. The source string is exhausted.
2. <result-identifier> becomes full.
3. An error occurs in the translation operation.

The <source-item-size> specifies the number of bits per item in <source-identifier>. <translate-item-size> specifies the bits per item in <translate-table> and <result-identifier>. The maximum length for <translate-item-size> and <source-item-size> is 24 bits. If the length of either <source-identifier> or <result-identifier> is not a multiple of its respective <translate-item-size>, the translation of the last item is undefined.

<translate-table> must be large enough to hold all items in <source-identifier>. Each item in <source-identifier> is used as a subscript into <translate-table> in order to determine the translated value. Refer to the B 1000 Systems SORT Reference Manual, form number 1090594, for complete information about the translation string.

SDL and UPL Syntax:

```
── TRANSLATE (<source-identifier>, <source-item-size>, ──────────────>
>──── <translate-table>, <translate-item-size>, <result-identifier> ─────┤
```

Syntax Semantics:

source-identifier
: This field can be any valid SDL/UPL identifier and specifies the source string for the TRANSLATE verb.

source-item-size
: This field can be any valid SDL/UPL literal, identifier, or expression that returns a binary value and specifies the number of bits per item in <source-identifier>.

translate-table
: This field can be any valid SDL/UPL literal, identifier, or expression that returns a binary value and specifies the table to use for translating <source-identifier> into the desired result.

translate-item-size
: This field can be any valid SDL/UPL literal, identifier, or expression that returns a value and specifies the number of bits per item in <translate-table> and <result-identifier>.

result-identifier
: This field can any valid SDL/UPL identifier and specifies the destination of the TRANSLATE verb.

TRANSLATE

Example:

```
DECLARE EBCDIC_TABLE  BIT (1024),
        ASCII_FIELD   BIT (70),
        EBCDIC_FIELD  BIT (80);

EBCDIC_TABLE := @000102033372D2E2F1605250B0C0D0E0F@
            CAT @101112133C3D322618193F271C1D1E1F@
            CAT @404F7F7B5B6C507D4D5D5C4E6B604B61@
            CAT @F0F1F2F3F4F5F6F7F8F97A5E4C7E6E6F@
            CAT @7CC1C2C3C4C5C6C7C8C9D1D2D3D4D5D6@
            CAT @D7D8D9E2E3E4E5E6E7E8E94AE05A5F6D@
            CAT @79818283848586878889919293949596@
            CAT @979899A2A3A4A5A6A7A8A9C06AD0A107@;

TRANSLATE (ASCII_FIELD, 7, EBCDIC_TABLE, 8, EBCDIC_FIELD);

% This example translates a USASCII-7 field into an EBCDIC field.
```

Example Program:

```
DECLARE   TRANSLATE_TABLE  BIT (128),
          ODT_INPUT        CHARACTER (20),
          ODT_OUTPUT       CHARACTER (40);

TRANSLATE_TABLE := @F0F1F2F3F4F5F6F7F8F9C1C2C3C4C5C6@;

DO FOREVER;
   DISPLAY ("ENTER ANY 20 CHARACTERS OR ENTER BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN DO;
                               DISPLAY ("GOOD BYE");
                               STOP;
                            END;

   TRANSLATE (ODT_INPUT, 4, TRANSLATE_TABLE, 8, ODT_OUTPUT);

   DISPLAY ("THE CHARACTERS ACCEPTED ARE EQUAL TO @" CAT ODT_OUTPUT
            CAT "@ IN HEXADECIMAL NOTATION");
END;
FINI;

% This example program accepts a 20-character field from the ODT
% and displays the hexadecimal value using the TRANSLATE verb.
% If BYE is entered, the program goes to end of job.
```

UNDO

The UNDO verb causes the program to exit a DO-group. Control is transferred to the statement immediately following the END statement for the corresponding DO-group.

A maximum of 16 nesting levels can be exited with the UNDO verb.

SDL and UPL Syntax:

```
── UNDO ──┬────────────────────┬── ; ──────────────────┤
          └── <identifier> ────┘
```

Syntax Semantics:

identifier
: This field can be any valid DO-group identifier and specifies the name of the DO-group to exit.

Examples:

```
UNDO;              % Causes the DO-group to be exited.

UNDO MAIN_LOOP;    % Causes the DO-group MAIN_LOOP to be exited.
```

Example Program:

Refer to the DO verb example program for an example program using the UNDO verb.

USE

The USE verb causes specific elements in a DEFINE statement to be declared in a procedure. This eliminates the need to declare all of the elements in a structure when only a portion are required. The name stack size is kept to a minimum and program maintenance is simplified. The SDL/UPL compiler generates the structure using fillers and the specified elements.

The USE verb must appear within a procedure and cannot appear on lexic level 0.

The referenced <defined-identifier> must define one structured DECLARE statement.

The structured DECLARE statement cannot contain arrays.

The DUMMY REMAPS keywords must be specified on the outermost level (01 level) of the structured DECLARE statement.

SDL and UPL Syntax:

```
               ┌──────── , ──────────────────────┐
── USE ( ──────┴──── <declared-identifier> ───────┴────── ) OF <defined-identifier>; ──────┤
```

Syntax Semantics:

declared-identifier
: This field can be any valid SDL/UPL identifier that is declared within a DEFINE statement.

define-identifier
: This field can be any valid SDL/UPL define identifier that defines a declaration statement which contains <declared-identifier>.

Example 1:

```
   DECLARE   PPB BIT (1440);              % The space is to be remapped.

   DEFINE PPB_DEC AS #                    % The DEFINE for the USE statement.

   DECLARE 01 DUMMY REMAPS PPB,           % The required DUMMY 01 level.
              03 PROG_NAME          CHARACTER (10),
              03 PROG_DATA_DICT     BIT (112),
              03 PROG_SEG_DICT      BIT (112),
              03 PROG_SORT_SPAD     BIT (28) #;

PROCEDURE GET_DICT;                                  % The procedure in
  USE (PROG_DATA_DICT, PROG_SEG_DICT) OF PPB_DEC;    % which the USE
                                                     % statement appears.
```

Example 2:

```
DEFINE X AS #
DECLARE 01 DUMMY REMAPS A,
           03 B              BIT (5),
              05 B1          BIT (2),
              05 B2          BIT (3),
           03 C              CHARACTER (10),
           03 D              BIT (1),
           03 E              FIXED,
           03 F              BIT (24) #;

PROCEDURE FIRST;
USE (C, D) OF X;
```

The following is the structure that the SDL/UPL compiler generates from the USE statement in procedure FIRST.

```
01 DUMMY REMAPS A,
   03 FILLER            BIT (5),
      05 FILLER         BIT (2),
      05 FILLER         BIT (3),
   03 C                 CHARACTER (10),
   03 D                 BIT (1),
   03 FILLER            FIXED,
   03 FILLER            BIT (24);
```

The keyword FILLER is substituted for the group identifier B. Normally, the SDL/UPL compiler generates a syntax error if FILLER is specified as the group-level identifier. This is allowed with the USE statement.

VALUE_DESCRIPTOR

The VALUE_DESCRIPTOR verb returns the descriptor of <address-field>. The value of an addressable item is represented by a descriptor on the top of the evaluation stack. When the VALUE_DESCRIPTOR verb is performed, this descriptor is placed on top of the value stack. The descriptor of the descriptor which is moved to value stack is placed on top of the evaluation stack with the NAME_VALUE STACK bit set.

SDL Syntax:

—— VALUE_DESCRIPTOR (<address-field>); ——|

Syntax Semantics:

identifier

This field can be any valid SDL identifier or expression that generates an address and specifies the name of the descriptor to be moved to the value stack.

Figure 9-6 shows the movement of the descriptor on the evaluation and value stacks when the VALUE_DESCRIPTOR verb is performed.

EVENT	EVALUATION STACK	VALUE STACK
BEFORE	(DESCRIPTOR OF X)	(VALUE OF X)
AFTER	(DESCRIPTOR OF DESCRIPTOR X)	(DESCRIPTOR OF X) (VALUE OF X)

G18324

Figure 9-6. Movement of Descriptor on Evaluation and Value Stacks

Example:

```
DECLARE ADDRESS BIT (24);
VALUE_DESCRIPTOR (ADDRESS);
```

VALUE_DESCRIPTOR

Example Program:

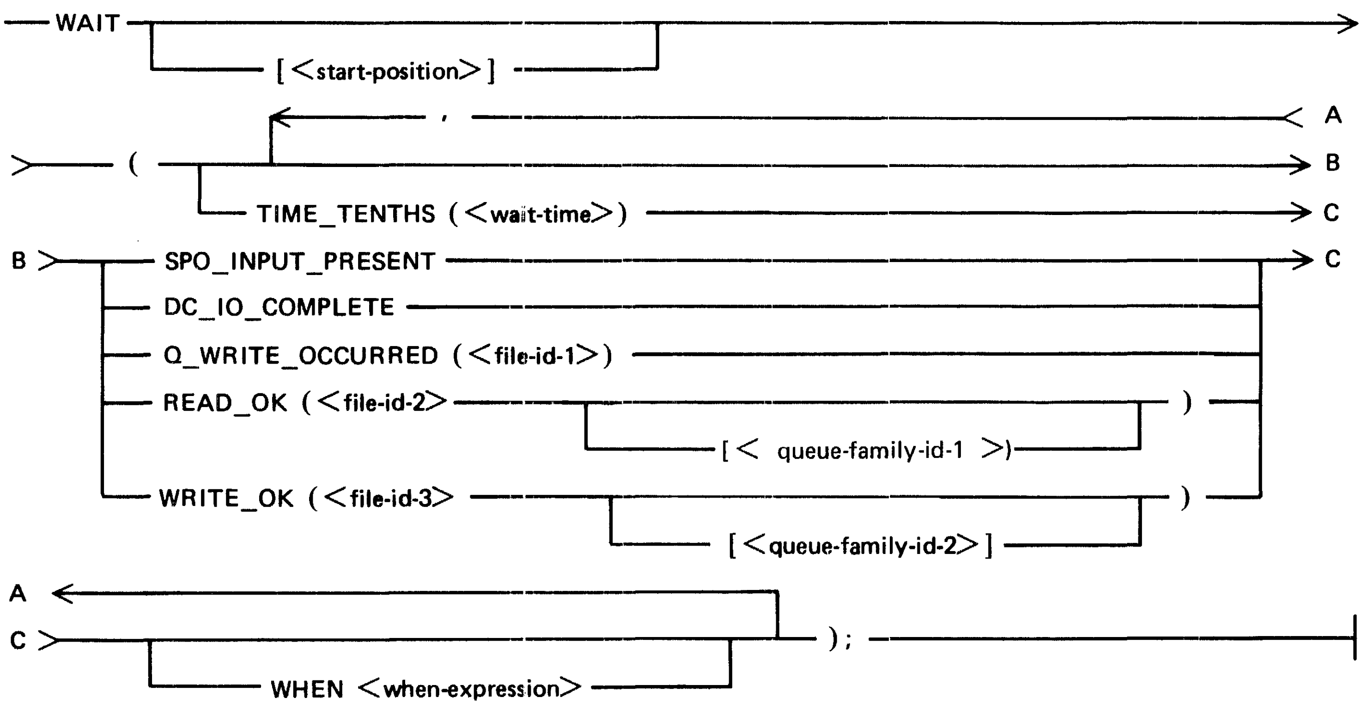
```
DECLARE  FIXED_FIELD  FIXED,
         BIT_FIELD    BIT (4),
         CHAR_FIELD   CHARACTER (1);

DISPLAY ("THE FOLLOWING IS THE DESCRIPTOR OF A FIXED FIELD:");
DISPLAY ("TYPE = " CAT
 CONVERT (SUBBIT (VALUE_DESCRIPTOR (FIXED_FIELD), 0, 8), CHARACTER));
DISPLAY ("LENGTH = " CAT
 CONVERT (SUBBIT (VALUE_DESCRIPTOR (FIXED_FIELD), 8, 16), CHARACTER));
DISPLAY ("ADDRESS = " CAT
 CONVERT (SUBBIT (VALUE_DESCRIPTOR (FIXED_FIELD), 24, 24), CHARACTER));

DISPLAY ("THE FOLLOWING IS THE DESCRIPTOR OF A BIT FIELD:");
DISPLAY ("TYPE = " CAT
 CONVERT (SUBBIT (VALUE_DESCRIPTOR (BIT_FIELD), 0, 8), CHARACTER));
DISPLAY ("LENGTH = " CAT
 CONVERT (SUBBIT (VALUE_DESCRIPTOR (BIT_FIELD), 8, 16), CHARACTER));
DISPLAY ("ADDRESS = " CAT
 CONVERT (SUBBIT (VALUE_DESCRIPTOR (BIT_FIELD), 24, 24), CHARACTER));

DISPLAY ("THE FOLLOWING IS THE DESCRIPTOR OF A CHARACTER FIELD:");
DISPLAY ("TYPE = " CAT
 CONVERT (SUBBIT (VALUE_DESCRIPTOR (CHAR_FIELD), 0, 8), CHARACTER));
DISPLAY ("LENGTH = " CAT
 CONVERT (SUBBIT (VALUE_DESCRIPTOR (CHAR_FIELD), 8, 16), CHARACTER));
DISPLAY ("ADDRESS = " CAT
 CONVERT (SUBBIT (VALUE_DESCRIPTOR (CHAR_FIELD), 24, 24), CHARACTER));

DISPLAY ("GOOD BYE");
STOP;
FINI;

% This example program displays the descriptor of fixed, bit, and
% character fields. The type, length, and address are displayed
% for each descriptor.
```

WAIT

The WAIT verb stops the program from processing until one of the events specified in the event list is TRUE. The WAIT verb returns a FIXED value which is the ordinal position (zero-relative) of the position in the event list of a TRUE event. Scanning begins with the ordinal position within the event list specified by <start-position> and continues to the last event. If no events are TRUE, the scanning continues with the first event in the event list until an event becomes TRUE. The events in the event list are identified by the keywords TIME_TENTHS, SPO_INPUT_PRESENT, DC_IO_COMPLETE, Q_WRITE_OCCURRED, READ_OK, and WRITE_OK. If <start-position> is not specified, <start-position> defaults to 0 (zero). If the value of <start-position> is greater than (number of events in the event list) minus 1, the MCP terminates the program with the following message.

INVALID COMPLEX.WAIT COMMUNICATE RECEIVED

SDL and UPL Syntax:



Syntax Semantics:

start-position

This field can be any valid integer, identifier, or expression that returns a binary value and specifies the ordinal position (zero relative) within the event list in which to begin the scanning for a TRUE event. If <start-position> is not specified, the value of <start-position> defaults to 0 (zero).

TIME_TENTHS

The keyword TIME_TENTHS is an event in the event list. The value of <wait-time> determines when the TIME_TENTHS event becomes TRUE. If the value of <wait-time> is equal to 0 (zero), the event is always TRUE. If TIME_TENTHS is specified in the event list, it must be the first event in the event list.

wait-time

This field can be any valid SDL/UPL integer, identifier, or expression that returns a binary value and specifies the length of time in tenths of a second to wait in order for the TIME_TENTHS event to become TRUE. The maximum value for <wait-time> is 864,000 (24 hours).

SPO_INPUT_PRESENT

The keyword SPO_INPUT_PRESENT is an event in the event list and becomes TRUE when a message from the operator at the ODT has been queued to the program.

DC_IO_COMPLETE

The keyword DC_IO_COMPLETE is an event in the event list and becomes TRUE when a previously initiated data communications read or write operation has been completed.

Q_WRITE_OCCURRED

The keyword Q_WRITE_OCCURRED is an event in the event list and becomes TRUE when a write operation has been performed by another program or process for the queue file specified by <file-id-1>.

file-id-1

This field can be any valid SDL/UPL queue file identifier that is opened INPUT or INPUT/OUTPUT and specifies the queue file identifier for the Q_WRITE_OCCURRED keyword.

READ_OK

The keyword READ_OK is an event in the event list and becomes TRUE when the buffer for the file specified by <file-id-2> contains a record waiting to be read.

file-id-2

This field can be any valid SDL/UPL file identifier that is opened INPUT or INPUT/OUTPUT and specifies the file for the READ_OK keyword.

If <file-id-2> is the file identifier for a queue file and <queue-family-id-1> is not specified, the READ_OK returns a TRUE condition even if there are no messages to read.

queue-family-id-1

This field can be any valid SDL/UPL identifier and specifies the subscript as the member of the queue file family. When the READ_OK becomes TRUE for a member within a queue file family, <queue-family-id-1> contains the value of the member within the queue file that has a record in the buffer to be read.

WRITE_OK

The keyword WRITE_OK is an event in the event list and becomes TRUE when the buffer for the file specified by <file-id-3> is empty and waiting for another write operation. If <queue-family-id-2> is specified, the WRITE_OK event applies to that queue family member.

file-id-3

This field can be any valid SDL/UPL file that is opened OUTPUT or INPUT/OUTPUT and specifies the file for the WRITE_OK keyword.

queue-family-id-2

This field can be any valid SDL/UPL identifier and specifies the subscript as the member of the queue file family.

WAIT

WHEN

The keyword WHEN causes an additional restriction of the occurrence of the associated event. If <when-expression> evaluates TRUE (rightmost bit equal to 1) and the associated event occurs, the event is TRUE. If <when-expression> evaluates FALSE (rightmost bit equal to 0) and the associated event is TRUE, the event is FALSE.

when-expression

This field can be any valid SDL/UPL identifier or expression and specifies the additional restriction for the WHEN keyword.

Example:

```
DECLARE EVENT FIXED,
        START FIXED;
EVENT := WAIT [START] (TIME_TENTHS (10),
                       SPO_INPUT_PRESENT,
                       Q_WRITE_OCCURRED (INQUEUE),
                       READ_OK (REMOTEFILE [STATION]),
                       WRITE_OK (TAPEFILE));
```

Example Program:

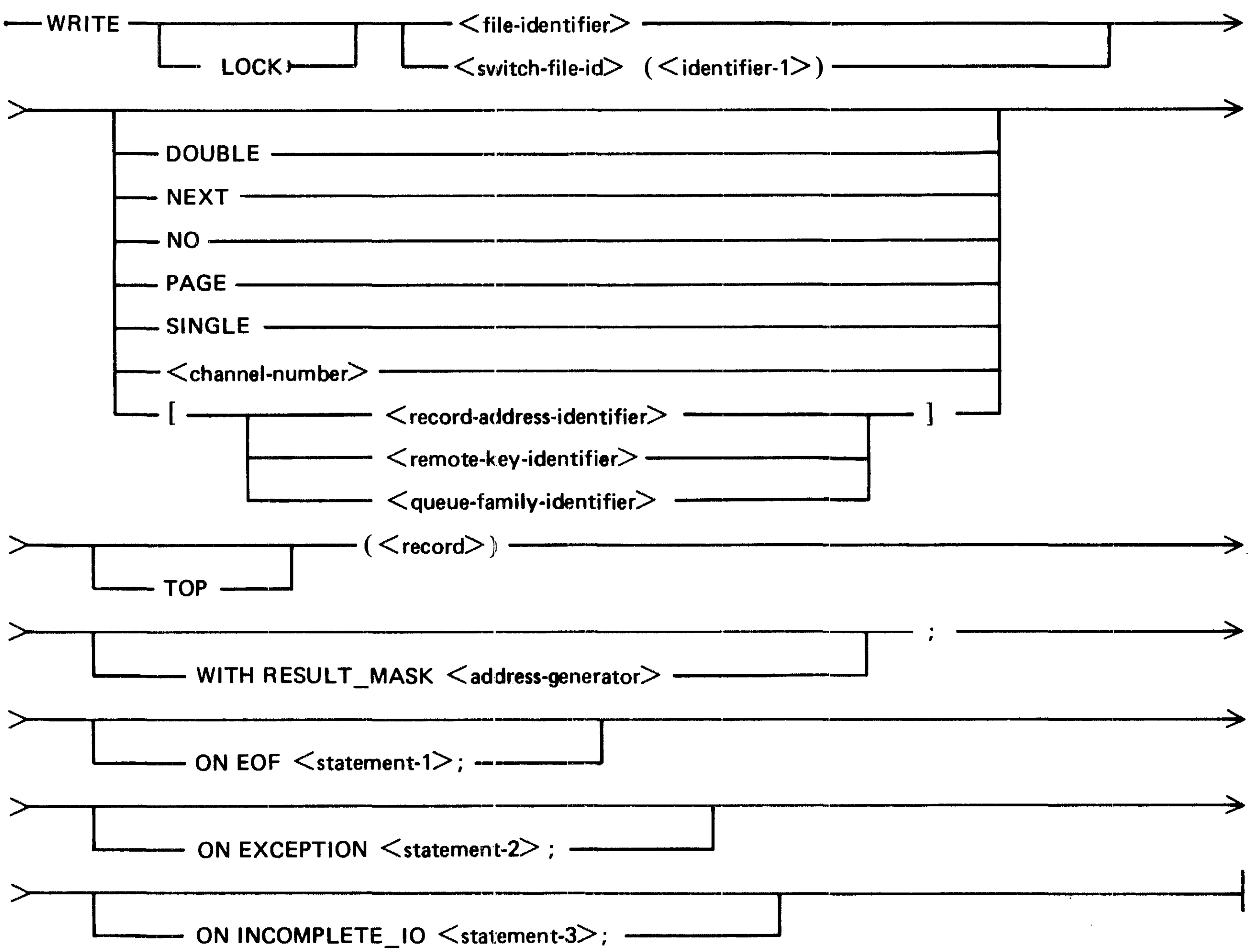
```
DECLARE ODT_INPUT        CHARACTER (30),
        START_POSITION   FIXED,
        EVENT            FIXED;

FILE    DISKFILE  (DEVICE = DISK,
                   RECORDS = 30/6);

DISPLAY ("THIS PROGRAM USES INPUT ACCEPT FROM THE ODT TO WRITE TO A"
     CAT " FILE CALLED DISKFILE.  ENTER BYE AT ANYTIME TO GO TO EOJ");
OPEN DISKFILE OUTPUT NEW;
START_POSITION := 1;
DO FOREVER;
  EVENT := WAIT [START_POSITION] (TIME_TENTHS (100), % WAIT 10 SECONDS
                                  SPO_INPUT_PRESENT,
                                  WRITE_OK (DISKFILE));

  CASE EVENT;
  /* 0 */  DISPLAY ("10 SECONDS HAVE PASSED SINCE LAST WRITE");
  /* 1 */  DO;
              ACCEPT ODT_INPUT;
              IF ODT_INPUT = "BYE" THEN DO;
                                          DISPLAY ("GOOD BYE");
                                          CLOSE DISKFILE LOCK;
                                          STOP;
                                       END;
              DISPLAY ("ODT INPUT ACCEPTED AND WRITE INITIATED");
              WRITE DISKFILE (ODT_INPUT);
           END;
  /* 2 */  DO FOREVER;
              DISPLAY ("OK TO WRITE -- ENTER DATA FOR WRITE");
              IF WAIT (TIME_TENTHS (100), SPO_INPUT_PRESENT) THEN UNDO;
           END;
  END CASE;
END;
FINI;

% This example program uses the WAIT verb to suspend the program
% until either 10 seconds have expired, the operator has queued
% a message to the program, or the buffer of DISKFILE is empty.
% If a message is queued to the program, the message is written
% to DISKFILE.  If BYE is entered the program goes to end of job.
```

WRITE

The WRITE verb causes the SDL/UPL program to write a record to the specified file.

The file attributes in the FILE declaration statement determine which of the position options (<record-address-identifier>, <remote-key-identifier>, <queue-family-identifier>, or carriage control keyword or <channel-number>) can be specified. <record-address-identifier> requires a file with a disk device type and random access or a card device type with the STACKERS open attribute specified at file open time. <remote-key-identifier> requires a file with a device type equal to REMOTE. <queue-family-identifier> requires two file attributes to be specified in the FILE declaration. A device type equal to QUEUE and the QUEUE_FAMILY_SIZE equal to the number of queue families are required. A device type equal to PRINTER is required for the carriage control position options.

SDL and UPL Syntax:



Syntax Semantics:

address-generator

This field can be any valid SDL/UPL address generator and specifies the name of the exception mask field.

LOCK
The keyword LOCK reserves a disk record for exclusive use of the program until a write operation is performed that does not specify LOCK.

file-identifier
This field can be any valid SDL/UPL file identifier that is declared in the FILE declarations and specifies the file in which the write operation is to take place.

switch-file-id
This field can be any valid SDL/UPL switch-file identifier that is declared in the FILE declarations and specifies the file in which the write operation is to take place.

identifier-1
This field can be any valid SDL/UPL identifier and specifies the switch file number.

DOUBLE
The keyword DOUBLE is used for files that are declared with a device type equal to PRINTER and causes the paper on the line printer to space forward two lines.

NEXT
The keyword NEXT is used for files that are declared with a device type equal to PRINTER and causes the paper on the line printer to skip to the next channel.

NO
The keyword NO is used for files that are declared with a device type equal to PRINTER and causes the paper on the line printer not to space forward.

PAGE
The keyword PAGE is used for files that are declared with a device type equal to PRINTER and causes the paper on the line printer to space to the top of page.

SINGLE
The keyword SINGLE is used for files that are declared with a device type equal to PRINTER and causes the paper on the line printer to space forward one line.

channel-number
This field can be any valid SDL/UDL integer and is used for files that are declared with a device type equal to PRINTER. < channel-number > specifies the channel number to advance to. The valid values for < channel-number > can be between 1 and 12, inclusive.

record-address-identifier
This field can be any valid SDL/UPL identifier and specifies the key location of a record within a file. <record-address-identifier> is valid for files with a device type equal to DISK RANDOM and DISK_PACK RANDOM. <record-address-identifier> is also valid for card files that are opened with the STACKERS open attribute.

<record-address-identifier> must be a binary value or an expression that returns a binary value. If the value is greater than 24 bits, only the rightmost 24 bits are used. For card files, the binary value of <record-address-identifier> must be less than or equal to 7, corresponding to a stacker available on the device. For example, if only two stackers are available on the card device, <record-address-identifier> equal to 3 is not valid.

WRITE

remote-key-identifier

This field can be any valid SDL/UPL identifier and specifies the relative station number (RSN) in the remote file to which the record is to be written.

<remote-key-identifier> is valid for files with a device type equal to REMOTE. The data type of <remote-key-identifier> must be equal to CHARACTER with a length of 10 bytes. The first three bytes (relative station number) of <remote-key-identifier> defaults to the character "001" if the maximum number of stations in the remote file is equal to 1. The maximum number of stations is specified in the FILE declarations. For example, specifying the following file attributes for a remote file causes the maximum number of stations for the remote file to be five.

(DEVICE = REMOTE, NUMBER_OF_STATIONS = 5, REMOTE_KEY)

Refer to the REMOTE_KEY file attribute for the format of <remote-key-identifier>.

queue-family-identifier

This field can be any valid SDL/UPL identifier and specifies the family number in the queue file in which to write the record.

<queue-family-identifier> is valid for files with a device type equal to QUEUE and with the QUEUE_FAMILY_SIZE greater than 1.

TOP

The keyword TOP is used for files that are declared with a device type equal to QUEUE and causes the record to be written at the front of the queue instead of at the tail. If a record is written at the front, a program that reads from the queue file reads this record.

record

This field can be any valid SDL/UPL literal, identifier, or expression that returns a value and specifies the data record to be written.

ON EOF

For printer files, the keywords ON EOF cause the program to perform <statement-1> if the end of page was encountered on the line printer. A printer file can take the ON EOF branch on reaching the end of page if the END_OF_PAGE_ACTION file attribute is specified in the FILE declaration statement.

For queue files, the keywords ON EOF cause the program to perform <statement-1> if the value of <queue-family-identifier> was out of range.

ON EXCEPTION

The keywords ON EXCEPTION cause the program to perform <statement-2> when an exception is encountered on the write operation and all the MCP retries are exhausted. For queue files, <statement-2> is performed when <queue-family-identifier> is out of range.

ON INCOMPLETE_IO

The keywords ON INCOMPLETE_IO cause the program to perform <statement-3>. For queue files, the INCOMPLETE_IO branch is performed when the number of records in the queue contains the value specified in QUEUE_MAX_MESSAGES file attribute. For other files, the INCOMPLETE_IO branch is performed when the write operation could not complete because the MCP had not physically completed writing the previous record. This occurs frequently with printer files.

statement-1
: This field can be any valid SDL/UPL statement and is performed when the ON EOF keywords, are specified in the WRITE statement for a printer file, and the end of the page is encountered on the line printer during the write operation. For queue files, if an exception occurs, the value for <queue-family-identifier> is out of range.

statement-2
: This field can be any valid SDL/UPL statement and is performed when the ON EXCEPTION keywords are specified, an exception is encountered, and the MCP has exhausted all the retries.

statement-3
: This field can be any valid SDL/UPL statement and is performed when the ON INCOMPLETE_IO keywords are specified for a queue file and the queue is full, or the write operation could not complete because the previous write operation was not complete.

WITH RESULT_MASK
: The keywords WITH RESULT_MASK cause the program to use <address-generator> as the exception mask identifier.

Variable-Length Records

The syntax of variable-length record write operations is identical to the syntax on fixed length records; however, the structure of the identifier and the value of the length field for the data differ from those for a fixed-length identifier.

Variable-length records are allowed only in tape and serial disk files that are declared with the file attribute VARIABLE. The RECORDS file attribute of the file must be large enough to hold the largest record to be written.

The first four bytes (characters) of the variable-length identifier contain record length information. On write operations, this record-size value must be included in the record.

The record length is equal to the number of bytes in the record plus the number of bytes in the record-size field (always 4). The record size is specified as a decimal value.

Example Program that Writes Variable-Length Records:

```
FILE PAYROLL (DEVICE = DISK, VARIABLE);

DECLARE 01 DISK_RECORD      CHARACTER(80),
           02 REC_SIZE      CHARACTER(4),
           02 DATA          CHARACTER(76),
        X                   REFERENCE;

DATA := "ABCDE";
REFER X TO DATA;
REDUCE X UNTIL LAST NEQ " ";
REC_SIZE := LENGTH(X) + 4;
WRITE PAYROLL (DISK_RECORD);
CLOSE PAYROLL LOCK;
STOP;
FINI;
```

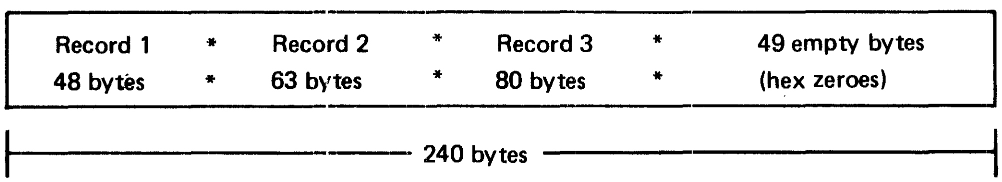
WRITE

To process variable-length records, the MCP builds a single buffer whose size is equal to the declared record size multiplied by the blocking factor. Variable-length records usually have a blocking factor equal to 1 (RECORDS = N/1). The MCP reads into its buffer as many complete logical records as it can. Logical records are not divided across physical record boundaries.

The following table shows example record numbers and associated record sizes in bytes. Assume the program specifies a record size equal to 240 bytes and the records and record sizes are:

Record Number	Data Record Size in Bytes (Including Record Size Field)
1	48
2	63
3	80
4	53
5	31

Figure 9-7 shows the contents of the 240-byte program buffer after a write operation is performed.



G18310

Figure 9-7. Contents of Program's Buffer After a Write Operation

Only records 1, 2, and 3 are written into the buffer because the next record (record 4) is too long to be stored in the remaining portion of the buffer. The unused portion of the buffer is filled with hexadecimal zeros.

Examples:

```
WRITE DISKFILE (FIELD);                               % Writes to the file
      ON EOF STOP;                                    % labeled DISKFILE.

WRITE DISK [INDEX] (FIELD);                           % Writes to the file
      ON EOF STOP;                                    % labeled DISK at
      ON EXCEPTION DISPLAY ("EXCEPTION");             % record address =
                                                      % the value of INDEX.

WRITE QUEUEFILE [NUMBER] (FIELD);                     % Writes to the file
      ON INCOMPLETE_IO DISPLAY ("QUEUE FULL");        % labeled QUEUEFILE
      ON EXCEPTION DISPLAY ("INVALID KEY");           % at queue family =
                                                      % the value of
                                                      % NUMBER.

WRITE REMOTEFILE [KEY] (FIELD);                       % Writes to the file
      ON EXCEPTION DISPLAY ("INVALID KEY");           % labeled REMOTEFILE
                                                      % at remote key = the
                                                      % value of KEY.
```

WRITE

Example Program:

```
DECLARE  ODT_INPUT   CHARACTER (30);

FILE  DISK  (DEVICE = DISK,
             RECORDS = 30/6),

      PRINT (DEVICE = PRINTER,
             RECORDS = 132/1),

      TAPE  (DEVICE = TAPE,
             RECORDS = 180/1),

      CARD  (DEVICE = PUNCH BACKUP DISK,
             RECORDS = 80/1);

OPEN DISK OUTPUT NEW;
OPEN PRINT OUTPUT NEW;
OPEN TAPE OUTPUT NEW;
OPEN CARD OUTPUT NEW;
DO MAIN_LOOP FOREVER;
   DISPLAY ("ENTER ANY 30 CHARACTERS FOR THE DATA RECORD OR ENTER"
            CAT " BYE FOR EOJ");
   ACCEPT ODT_INPUT;
   IF ODT_INPUT = "BYE" THEN UNDO MAIN_LOOP;

   WRITE DISK (ODT_INPUT);
     ON EXCEPTION DO;
                     DISPLAY ("EXCEPTION ENCOUNTERED ON DISK WRITE");
                     UNDO MAIN_LOOP;
                  END;

   WRITE PRINT (ODT_INPUT);
     ON EXCEPTION DO;
                     DISPLAY ("EXCEPTION ENCOUNTERED ON PRINT WRITE");
                     UNDO MAIN_LOOP;
                  END;

   WRITE TAPE (ODT_INPUT);
     ON EXCEPTION DO;
                     DISPLAY ("EXCEPTION ENCOUNTERED ON TAPE WRITE");
                     UNDO MAIN_LOOP;
                  END;
```

WRITE

```
    WRITE CARD (ODT_INPUT);
      ON EXCEPTION DO;
                            DISPLAY ("EXCEPTION ENCOUNTERED ON CARD WRITE");
                            UNDO MAIN_LOOP;
                         END;

 END MAIN_LOOP;
 DISPLAY ("GOOD BYE");
 CLOSE DISK RELEASE;
 CLOSE PRINT RELEASE;
 CLOSE TAPE RELEASE;
 CLOSE CARD RELEASE;
 STOP;
 FINI;

 % This example program accepts input from the ODT and uses the
 % WRITE verb to write to a disk, printer, tape, and card file.
 % If BYE is entered, the program goes to end of job.
```

WRITE__FILE__HEADER

The WRITE__FILE__HEADER verb writes the disk file header information for the file specified by <file-identifier>. This verb is intended only for use in B 1000 system software, and extreme caution is advised when writing disk file header information.

SDL Syntax:

```
—— WRITE_FILE_HEADER (<file-identifier>, <destination>); ——————————————>

>——┬——/1\——— ON FILE_MISSING <statement-1>; ———————┬———————————————|
   └——/1\——— ON FILE_LOCKED <statement-2>; ————————┘
```

Syntax Semantics:

file-identifier
: This field can be any valid SDL literal, identifier, or expression that returns a character value with a CHARACTER data type and specifies the name of the file. <file-identifier> is expected to be a 30-character value, where the first 10 characters are the pack identifier, the second 10 characters are the multifile identifier, and the third 10 characters are the file identifier. Each of the file identifiers is left-justified in their respective fields. If only one file name exists, the file name is left-justified in the second 10 characters of the file name and the first and third 10 characters are set to blank.

destination
: This field can be any valid SDL identifier and specifies the receiving field for the disk-file-header information. This field is expected to be from 576 to 4320 bits in length, depending upon the number of disk areas allocated for the file.

ON FILE__MISSING
: The keywords ON FILE__MISSING cause <statement-1> to be performed if the file name specified by <file-identifier> is not in the disk directory.

ON FILE__LOCKED
: The keywords ON FILE__LOCKED cause <statement-2> to be performed if the file name specified by <file-identifier> is opened by another program with the LOCK open option set.

statement-1
: This field can be any valid SDL statement and is performed if the keywords ON FILE__MISSING are specified and <file-identifier> is not in the disk directory.

statement-2
: This field can be any valid SDL statement and is performed if the keywords ON FILE__LOCKED are specified and <file-identifier> is currently opened with the LOCK open option set.

WRITE_FILE_HEADER

Example:

```
DECLARE   DISKFILE   CHARACTER (30),        % The disk file header for
          SOURCE     BIT (4320);            % the file identifier
DISKFILE :=                                 % USER/MASTER/FILE is
  "USER       MASTER     FILE       ";      % written using the
READ_FILE_HEADER (DISKFILE, SOURCE);        % information in identifier
 ON FILE_MISSING STOP;                      % SOURCE.
 ON FILE_LOCKED STOP;
```

Example Program:

```
DECLARE FILENAME       CHARACTER (30),
        DESTINATION    BIT (4320),
        SOURCE         BIT (4320),
        DFH_LENGTH     BIT (16);

DO MAIN_LOOP FOREVER;
   DISPLAY ("ENTER THE 30 CHARACTER FILE NAME LEFT JUSTIFIED OR ENTER "
            CAT "BYE TO GO TO EOJ");
   ACCEPT FILENAME;
   IF FILENAME = "BYE" THEN DO;
                            DISPLAY ("GOOD BYE");
                            STOP;
                         END;
   DO READ_DFH;
      READ_FILE_HEADER (FILENAME, DESTINATION);
        ON FILE_MISSING DO;
                         DISPLAY ("FILE " CAT FILENAME CAT
                                  "NOT IN THE DISK DIRECTORY");
                         UNDO READ_DFH;
                      END;
        ON FILE_LOCKED DO;
                        DISPLAY ("FILE " CAT FILENAME CAT
                                 " IS LOCKED");
                        UNDO READ_DFH;
                     END;
   END READ_DFH;
   DFH_LENGTH := SUBBIT (DESTINATION, 91, 16);
   SOURCE := DESTINATION;

   WRITE_FILE_HEADER (FILENAME, SUBBIT (SOURCE, 0, DFH_LENGTH));

   DISPLAY ("THE FOLLOWING DISK FILE HEADER INFORMATION WAS WRITTEN");
   DISPLAY (CONVERT (SUBBIT (SOURCE, 0, DFH_LENGTH), CHARACTER));
END MAIN_LOOP;
FINI;

% This example program accepts from the ODT a 30-character file name
% and rewrites the disk file header information on top of the
% existing disk file header. If BYE is entered, the program goes
% to end of job.
```

WRITE__FPB

The WRITE__FPB verb writes the file parameter block (FPB) of the file indicated by <file-identifier> or <file-number> and uses the FPB information stored in <source>.

SDL Syntax:

```
── WRITE_FPB ( ──┬── <file-identifier> ──┬── , <source>) ; ──┤
                 └── <file-number> ──────┘
```

Syntax Semantics:

file-identifier
: This field can be any valid SDL file identifier and specifies the file name from which to write the FPB information.

file-number
: This field can be any valid SDL switch file number and specifies the file number within the program from which to write the FPB information.

source
: This field can be any valid SDL identifier and specifies the name of the field used to obtain the FPB information. The length of this field must be 2096 bits.

Example:

```
DECLARE FPB_INFO  BIT (1440);      % The file parameter block
WRITE_FPB (DISKFILE, FPB_INFO);    % information of the file DISKFILE
                                   % is stored into identifier
                                   % FPB_INFO.
```

Example Program:

Refer to the READ__FPB verb for an example program using the WRITE__FPB verb.

WRITE_OVERLAY

The WRITE_OVERLAY verb writes to the disk address specified in <overlay-information> and uses the data segment beginning and ending addresses specified in <overlay-information>.

The WRITE_OVERLAY verb is used by the SDL intrinsics.

SDL Syntax:

```
——WRITE_OVERLAY (<overlay-information>); ————————————————————————|
```

Syntax Semantics:

overlay-information

This field can be any valid SDL literal, identifier, or expression that returns a 76-bit value and has the following format.

Bits	Description
0-3	EU = 0 (not used)
4-27	Base-relative beginning address
28-51	Base-relative ending address
52-75	Disk address, relative to program area.

Example:

```
DECLARE 01 OVERLAY_RECORD   BIT (76),  % The data segment at disk
           03 EU            BIT (4),   % address @008A7E@ is
           03 BEGIN_ADDR    BIT (24),  % stored in the program's
           03 END_ADDR      BIT (24),  % base-to-limit area
           03 DISK_ADDR     BIT (24);  % beginning at @71E7A2@
EU := 1;                               % and ending at @71F8420@.
BEGIN_ADDR := @71E7A2@;
END_ADDR := @71F842@;
DISK_ADDR := @008A78@;
WRITE_OVERLAY (OVERLAY_RECORD);
```

X_ADD

The X_ADD verb causes the add operation to be performed with <expression-1> and <expression-2>. <expression-1> and <expression-2> are treated as bit strings and the full length of each is used, not just the rightmost 24 bits.

If <expression-1> or <expression-2> are different lengths, the shorter is padded on the left with binary zeros. The length of the sum is equal to the length of the longer of <expression-1> or <expression-2>.

SDL and UPL Syntax:

```
── X_ADD ( <expression-1>, <expression-2> ) ──────────────────────────┤
```

Syntax Semantics:

expression-1
: This field can be any valid SDL/UPL expression and specifies the first operand for the extended arithmetic add operation.

expression-2
: This field can be any valid SDL/UPL expression and specifies the second operand for the extended arithmetic add operation.

Examples:

```
X := X_ADD (@13AFCHK@, @2374@);

X := X_ADD (TIMER, (TIMER - 1000));
```

X_DIV

The X_DIV verb causes the divide operation to be performed with <expression-1> and <expression-2>. <expression-1> is divided by <expression-2>. <expression-1> and <expression-2> are treated as bit strings and the full length of each is used, not just the rightmost 24 bits.

The length of the quotient is the length of <expression-1>.

SDL and UPL Syntax:

```
—— X_DIV (<expression-1>, <expression-2> ) ——————————————————————————|
```

Syntax Semantics:

expression-1
: This field can be any valid SDL/UPL expression and specifies the first operand for the extended arithmetic divide operation.

expression-2
: This field can be any valid SDL/UPL expression and specifies the second operand for the extended arithmetic divide operation.

Examples:

```
X := X_DIV (@56AFGHK@, @2374@);

X := X_DIV (TIMER, (TIMER - 1000));
```

X_MOD

The X_MOD verb causes the modulo operation to be performed with <expression-1> and <expression-2>. <expression-2> is the modulus. <expression-1> and <expression-2> are treated as bit strings and the full length of each is used, not just the rightmost 24 bits.

The length of the residue is the length of <expression-1>.

SDL and UPL Syntax:

```
—— X_MOD (<expression-1>, <expression-2>) ——————————————————————————|
```

Syntax Semantics:

expression-1
: This field can be any valid SDL/UPL expression and specifies the first operand for the extended arithmetic modulo operation.

expression-2
: This field can be any valid SDL/UPL expression and specifies the second operand for the extended arithmetic modulo operation.

Examples:

```
X := X_MOD (@12345FK@, @2374@);

X := X_MOD (TIMER, (TIMER - 1000));
```

X_MUL

The X_MUL verb causes the multiply operation to be performed with <expression-1> and <expression-2>. <expression-1> and <expression-2> are treated as bit strings and the full length of each is used, not just the rightmost 24 bits.

The length of the product is the sum of the lengths of <expression-1> and <expression-2>. This sum cannot exceed 65,535 bits.

SDL and UPL Syntax:

```
—— X_MUL (<expression-1>, <expression-2>) ——————————————————————————————|
```

Syntax Semantics:

expression-1
: This field can be any valid SDL/UPL expression and specifies the first operand for the extended arithmetic multiply operation.

expression-2
: This field can be any valid SDL/UPL expression and specifies the second operand for the extended arithmetic multiply operation.

Examples:

```
X := X_MUL (@45HK@, @23743@);

X := X_MUL (TIMER, (TIMER - 1000));
```

X_SUB

The X_SUB verb causes the subtraction operation to be performed with <expression-1> and <expression-2>. <expression-1> and <expression-2> are treated as bit strings and the full length of each is used, not just the rightmost 24 bits.

If <expression-1> and <expression-2> are of different lengths, the shorter is padded on the left with binary zeros. The length of the difference is equal to the length of the longer of <expression-1> or <expression-2>.

SDL and UPL Syntax:

```
—— X_SUB (<expression-1>, <expression-2>) ——————————————————————————|
```

Syntax Semantics:

expression-1
: This field can be any valid SDL/UPL expression and specifies the first operand for the extended arithmetic subtraction operation.

expression-2
: This field can be any valid SDL/UPL expression and specifies the second operand for the extended arithmetic subtraction operation.

Examples:

```
X := X_SUB (@89FFFF@, @2374@);

X := X_SUB (TIMER, (TIMER - 1000));
```

ZIP

The ZIP verb passes control information to the MCP.

SDL and UPL Syntax:

```
── ZIP <MCP-command> ; ──────────────────────────────────────┤
```

Syntax Semantics:

MCP-command
This field can be any valid SDL/UPL literal, identifier, or expression that returns a value and specifies a valid MCP control statement as defined in the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982.

Examples:

```
ZIP "SC OPEN";                        % Sets the OPEN option in the MCP.

ZIP "EX DMPALL";                      % Begins the execution of the
                                      % DMPALL program.

ZIP "COMPILE PRINT UPL SYNTAX";       % Program PRINT is to be compiled
                                      % for syntax only.

ZIP "SV LPA";                         % The MCP is requested to reserve
                                      % line printer LPA.
```

SECTION 10
COMPILER OPTIONS AND PASSES

This section describes the compiler options and the conditional compilation facility available in the SDL/UPL compiler. Additionally, a brief description of the function of the four passes of the SDL/UPL compiler is presented.

COMPILE DECK

The compile deck is a card file that contains the MCP control commands and the SDL/UPL source program.

To compile an SDL/UPL program from cards, the following control cards are required:

```
?COMPILE <program-name> WITH UPL LIBRARY
?DATA CARDS
   •
   •
   <SDL/UPL source cards>
   •
   •
?END
```

To compile an SDL/UPL program from a disk file, the following control information is required:

```
?COMPILE <program-name> WITH UPL LIBRARY;
?FILE CARDS NAME <disk-file-name> DISK DEFAULT;
```

SDL/UPL COMPILER FILES

The following are the files used by the SDL/UPL compiler.

File	Description
CARDS	Input file to read source records.
SOURCE	Primary source if the $MERGE compiler-directing option is specified.
NEWSOURCE	Updated source output file if the $NEW compiler-directing option is specified.
LINE	Line printer file used to print the compile source listing.
ERROR.LINE	Line printer file used to print errors generated during the compile.

The $NEW compiler-directing option creates a source file on disk that can have other source images merged during compilations.

Example

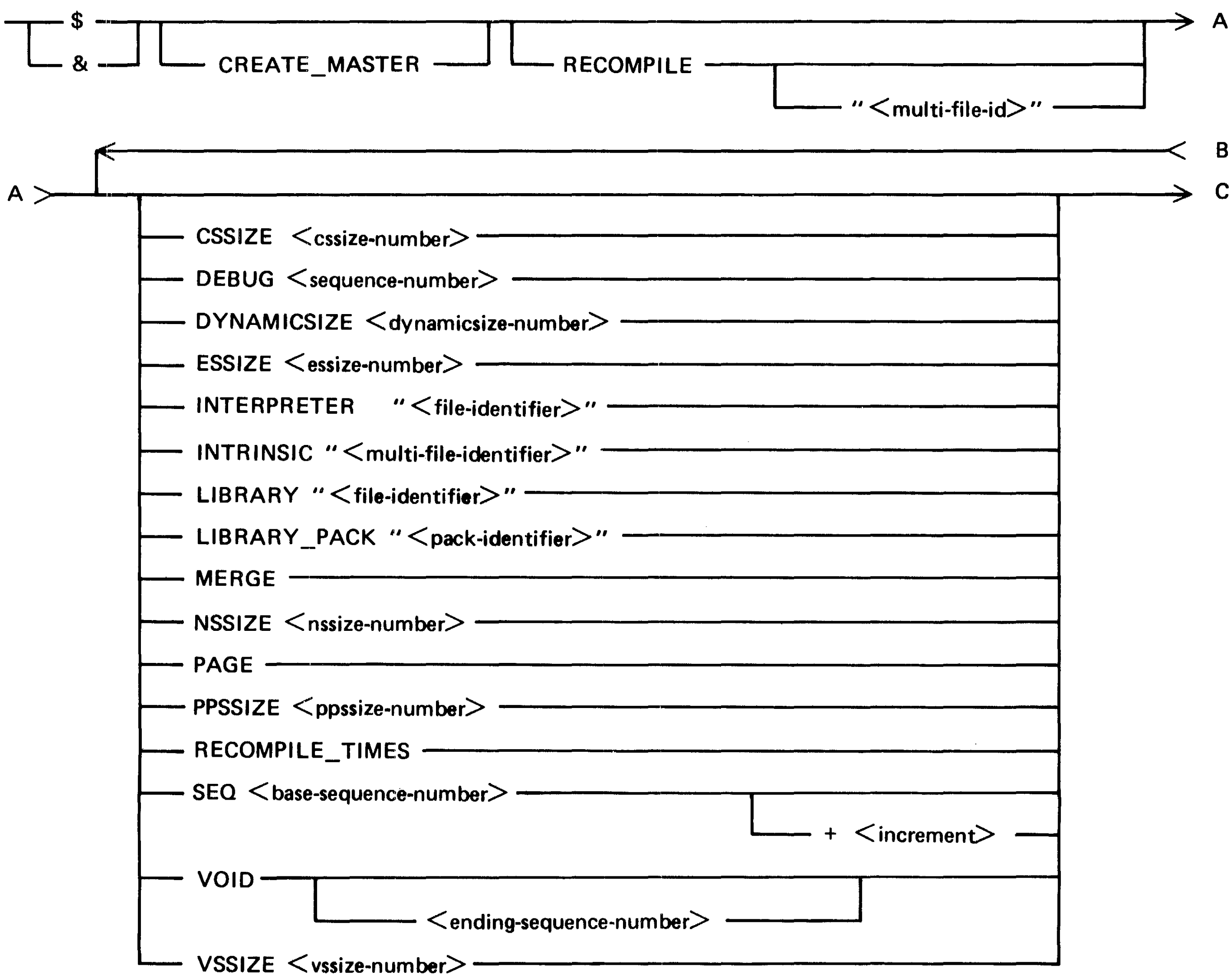
To compile using a source file on disk and to merge additional source images, use the following control information.

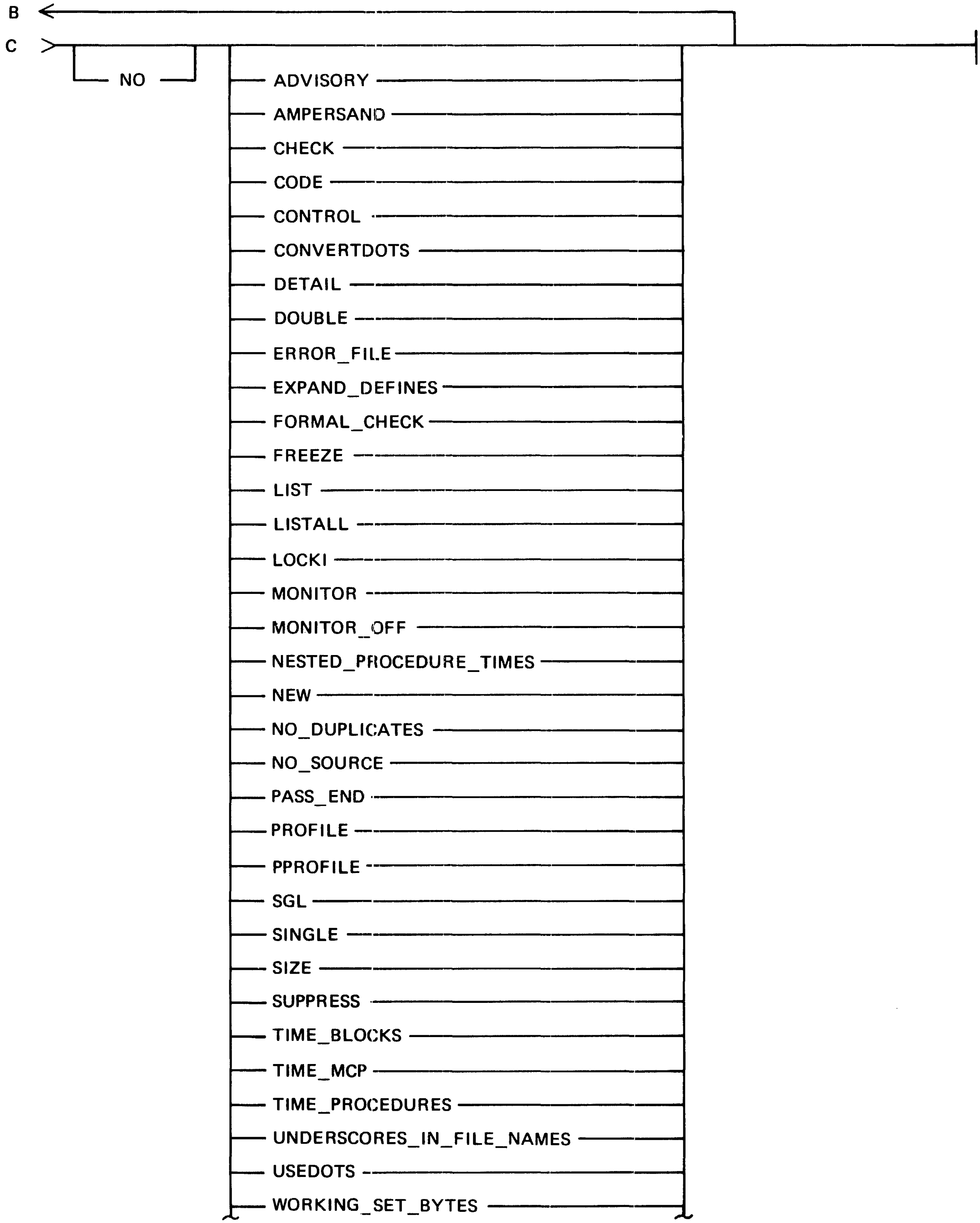
```
?COMPILE <program-name> WITH UPL LIBRARY
 <file statement for SOURCE file>
 <file statement for NEWSOURCE file>
?DATA CARDS
 $ MERGE
 $ NEW
    .
    .
    <UPL source images to be merged>
    .
    .
    FINI
?END
```

COMPILER-DIRECTING OPTIONS

All compiler option control records must have an ampersand (&) or dollar sign ($) character in position 1. The keywords can appear anywhere from positions 2 through 72 and must be separated by a blank character. Positions 73 through 80 are reserved for the sequence numbers.

SDL Syntax:

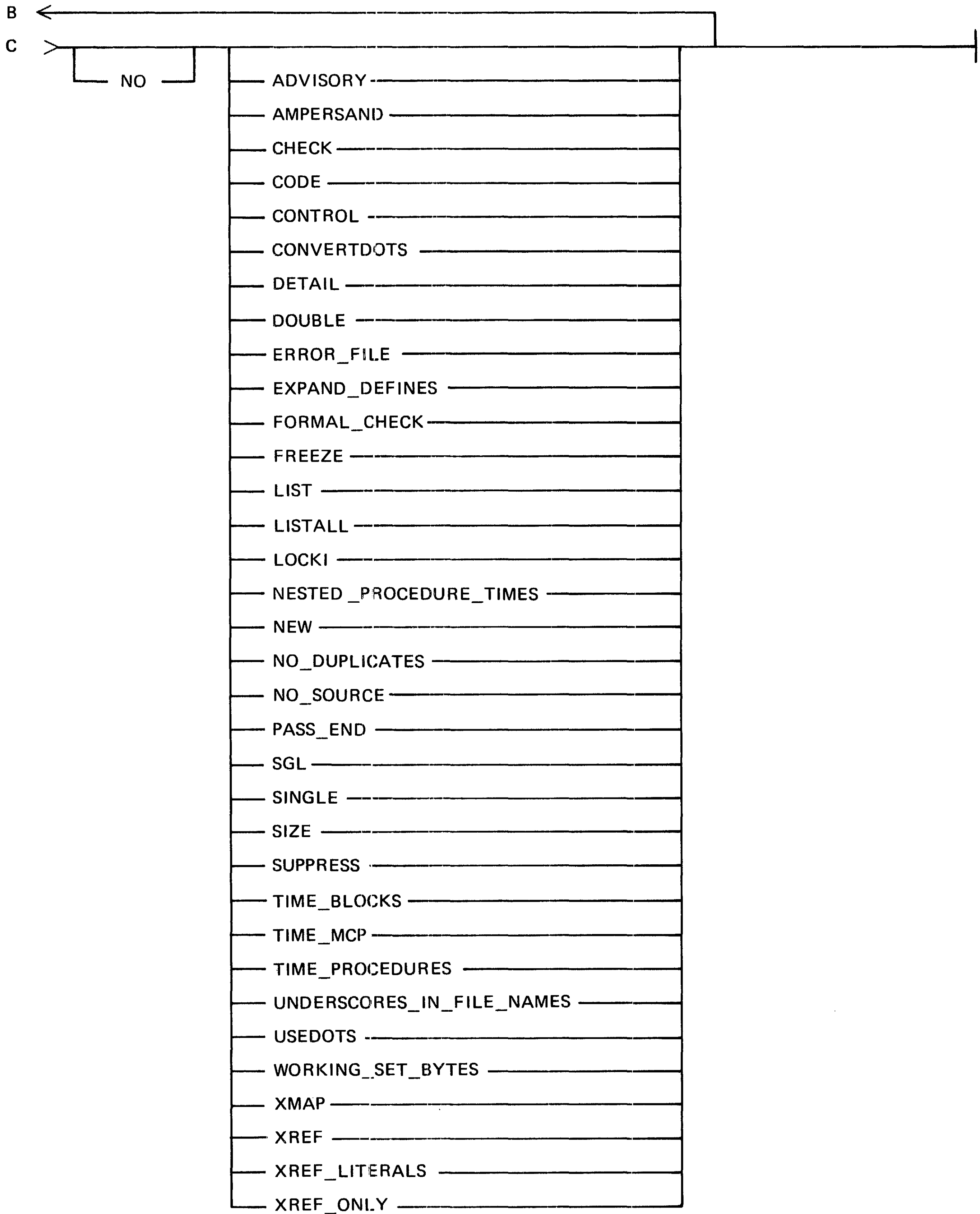



B
C
NO
ADVISORY
AMPERSAND
CHECK
CODE
CONTROL
CONVERTDOTS
DETAIL
DOUBLE
ERROR_FILE
EXPAND_DEFINES
FORMAL_CHECK
FREEZE
LIST
LISTALL
LOCKI
MONITOR
MONITOR_OFF
NESTED_PROCEDURE_TIMES
NEW
NO_DUPLICATES
NO_SOURCE
PASS_END
PROFILE
PPROFILE
SGL
SINGLE
SIZE
SUPPRESS
TIME_BLOCKS
TIME_MCP
TIME_PROCEDURES
UNDERSCORES_IN_FILE_NAMES
USEDOTS
WORKING_SET_BYTES

```
  ├── XMAP ──────────────────┤
  ├── XREF ──────────────────┤
  ├── XREF_LITERALS ─────────┤
  └── XREF_ONLY ─────────────┘
```

UPL Syntax:

```
──┬─ $ ─┬─┬───────────────────┬─┬─────────────────────────────────────┬──> A
  └─ & ─┘ └── CREATE_MASTER ──┘ └── RECOMPILE ──┬─────────────────────┤
                                                └── "<multi-file-id>" ┘

      ┌<──────────────────────────────────────────────────────────────< B
A >───┴──┬────────────────────────────────────────────────────┬───────> C
         ├── CSSIZE <cssize-number> ──────────────────────────┤
         ├── DEBUG <sequence-number> ─────────────────────────┤
         ├── DYNAMICSIZE <dynamicsize-number> ────────────────┤
         ├── ESSIZE <essize-number> ──────────────────────────┤
         ├── INTERPRETER "<file-identifier>" ─────────────────┤
         ├── INTRINSIC "<multi-file-identifier>" ─────────────┤
         ├── LIBRARY "<file-identifier>" ─────────────────────┤
         ├── LIBRARY_PACK "<pack-identifier>" ────────────────┤
         ├── MERGE ───────────────────────────────────────────┤
         ├── NSSIZE <nssize-number> ──────────────────────────┤
         ├── PAGE ────────────────────────────────────────────┤
         ├── PPSSIZE <ppssize-number> ────────────────────────┤
         ├── RECOMPILE_TIMES ─────────────────────────────────┤
         ├── SEQ <base-sequence-number> ───────┬──────────────┤
         │                                     └ + <increment>┤
         ├── VOID ──┬──────────────────────────────┬──────────┤
         │          └─ <ending-sequence-number> ───┘          │
         └── VSSIZE <vssize-number> ──────────────────────────┘
```


B
C
NO
ADVISORY
AMPERSAND
CHECK
CODE
CONTROL
CONVERTDOTS
DETAIL
DOUBLE
ERROR_FILE
EXPAND_DEFINES
FORMAL_CHECK
FREEZE
LIST
LISTALL
LOCKI
NESTED_PROCEDURE_TIMES
NEW
NO_DUPLICATES
NO_SOURCE
PASS_END
SGL
SINGLE
SIZE
SUPPRESS
TIME_BLOCKS
TIME_MCP
TIME_PROCEDURES
UNDERSCORES_IN_FILE_NAMES
USEDOTS
WORKING_SET_BYTES
XMAP
XREF
XREF_LITERALS
XREF_ONLY

Syntax Semantics:

$

The dollar sign ($) character must be specified in the first position of the control record. If a new source file is to be generated by the SDL/UPL compiler, the dollar sign ($) character in this control record causes the control record to be excluded in the new source file which is labeled NEWSOURCE.

&

The ampersand (&) character must be specified in the first position of the control record. If a new source file is to be generated by the SDL/UPL compiler, the ampersand (&) character causes this control record to be included in the new source file.

ADVISORY

The keyword ADVISORY causes the SDL/UPL compiler to include advisory messages in the program listing. The default is to include advisory messages.

AMPERSAND

The keyword AMPERSAND causes the SDL/UPL compiler to include the control records that contain the ampersand (&) character in the first position.

base-sequence-number

This field can be any 8-digit number and specifies the sequence number where resequencing of the source file is to begin. The field is used in conjunction with the SEQ keyword and defaults to 1000.

CHECK

The keyword CHECK causes the SDL/UPL compiler to check the source file for sequence errors.

CODE

The keyword CODE causes the SDL/UPL compiler to list the generated S-machine code in the program listing.

CONTROL

The keyword CONTROL causes the SDL/UPL compiler to list the compiler control record in the program listing.

CONVERTDOTS

The keyword CONVERTDOTS causes all the period (.) characters to be converted to underscore (_) characters for all the SDL/UPL compiler output files. This control option does not change the period (.) character in file identifiers to the underscore (_) character.

CREATE_MASTER

The keyword CREATE_MASTER causes the master information file to be created for subsequent partial compilation. This control option must be specified in the first record in the source file.

The XMAP compiler option is not allowed when the CREATE_MASTER compiler option is specified.

CSSIZE

The keyword CSSIZE causes the control stack to be changed to the value specified by <cssize-number>. The SDL/UPL compiler determines the default control stack size used for each program based on standard algorithms.

cssize-number
This field can contain any number and specifies the number of entries in the control stack.

DEBUG
The keyword DEBUG is only for use in debugging the SDL/UPL compiler.

DETAIL
The keyword DETAIL causes all define identifiers used in the SDL/UPL program to be expanded in the program listing.

DOUBLE
The keyword DOUBLE causes the program listing to be double spaced.

DYNAMICSIZE
The keyword DYNAMICSIZE causes the amount of dynamic memory (in bits) specified by <dynamicsize-number> to be used for paged-array pages. The SDL/UPL compiler generates a default value based on standard algorithms.

dynamicsize-number
This field can contain any number and specifies the amount of dynamic memory in bits to use for paged-array pages.

ending-sequence-number
This field can be any 8-digit number and specifies the upper-bound sequence number of the source records to be excluded in the new source file and compilation of the program. This field is used in conjunction with the VOID keyword.

ERROR_FILE
The keyword ERROR_FILE causes a separate file to be created: this file contains only syntax errors and warning messages for applicable source images generated during the compilation of the SDL/UPL program.

ESSIZE
The keyword ESSIZE causes the number specified by <essize-number> to be used for the evaluation stack size. The SDL/UPL compiler determines the default evaluation stack size used for each program based on standard algorithms.

essize-number
This field can contain any number and specifies the number of entries allowed in the evaluation stack.

EXPAND_DEFINES
The EXPAND_DEFINES keyword causes all identifiers used in define identifiers to be included in the cross-reference file. This keyword is used in conjunction with the compiler options XREF and XREF_ONLY.

file-identifier
This field can be any file identifier that follows the B 1000 file-naming convention and specifies the file name of the interpreter or the file name of a library file.

FORMAL_CHECK
The keyword FORMAL_CHECK causes the actual parameters and values passed to or returned from procedures to be checked against their corresponding formal parameters and procedure formal types.

FREEZE
The keyword FREEZE causes the freeze bit to be set in the object of the File Parameter Block (FPB) of the program, and prevents the run structure nucleus of the program from being rolled out to disk during execution.

increment
This field can contain any number and specifies the number with which to increment the sequence number. This field is used in conjunction with the SEQ keyword and defaults to 1000.

INTERPRETER
The keyword INTERPRETER changes the name of the interpreter to the name specified by <file-identifier>. The default interpreter name is SDL/INTERP1S.

INTRINSIC
The keyword INTRINSIC changes the multifile identifier of the intrinsic files that are to be used. The default multifile identifier is SDL.INTRIN.

LIBRARY
The keyword LIBRARY causes the SDL/UPL compiler to include the source records in the file specified by <file-identifier> in the compilation of the program.

LIBRARY_PACK
The keyword LIBRARY_PACK causes the SDL/UPL compiler to expect all library files to be on the disk pack specified by <pack-identifier>.

LIST
The keyword LIST causes the program listing to be created. The default is to create the program listing.

LISTALL
The keyword LISTALL causes all of the source file to be listed in the program listing, whether or not it was conditionally excluded. Specifying LISTALL turns on LIST while NO LISTALL does not turn off LIST. To turn both options off, specify NO LIST.

LOCKI
The keyword LOCKI causes the intermediate work files of the SDL/UPL compiler to be locked in the disk directory as they are created.

MERGE
The keyword MERGE specifies that the primary source file is in a tape or disk file labeled SOURCE and the secondary or merging file is a card file labeled CARDS. The card file is merged with the tape or disk file based on the sequence number of the input records.

MONITOR
The keyword MONITOR causes the run-time tracing of procedure calls to be invoked.

MONITOR_OFF
The keyword MONITOR_OFF causes the MONITOR option to be reset.

multi-file-id
This field can be any multifile identifier that follows the B 1000 file-naming convention.

NEW

The keyword NEW causes a new source file labeled NEWSOURCE to be created.

NO

The keyword NO turns off the applicable compiler option that follows the keyword NO.

NO_DUPLICATES

The keyword NO_DUPLICATES causes the SDL/UPL compiler to suppress the check for unique identifiers in the source file. This reduces the amount of time needed to compile the program.

NO_SOURCE

The keyword NO_SOURCE causes the SDL/UPL compiler to suppress creation of a program listing. This option shortens the size of the SDL/UPL work files and decreases the compile time.

NSSIZE

The keyword NSSIZE causes the number specified by <nssize-number> to be used as the name stack size. The SDL/UPL compiler generates the default name stack size used by each program based on standard algorithms.

nssize-number

This field can contain any number and specifies the number of entries allowed in the name stack.

pack-identifier

This field can be any valid pack identifier that follows the B 1000 file-naming convention and specifies the disk pack name for library files.

PAGE

The keyword PAGE causes the SDL/UPL compiler to continue printing the program listing on the top of a new page.

PASS_END

The keyword PASS_END causes the SDL/UPL compiler to display the total number of syntax errors that have been generated and the total elapsed processor time at the end of each pass of the compiler.

PPSSIZE

The keyword PPSSIZE causes the number specified by <ppssize-number> to be used as the program pointer stack size. The SDL/UPL compiler generates the default program pointer stack used by each program based on standard algorithms.

ppssize-number

This field can contain any number and specifies the number of entries allowed in the program pointer stack.

PROFILE

The keyword PROFILE causes a dynamic array to be generated. Each element in the array is a counter of the number of times that a transfer of control statement (DO-group, IF statement, or CASE statement) is performed. An index into the array appears in the program listing following the statement in which control is transferred. The statements with the highest counter value are the most used statements.

PPROFILE

The keyword PPROFILE causes a dynamic array to be generated. Each element in the array is a counter of the number of times that a procedure is entered. An index into the array appears in the program listing following the procedure declaration. The procedures with the highest counter value are the most used procedures.

RECOMPILE

The keyword RECOMPILE invokes the partial compilation facility of the SDL/UPL compiler using master information files from a previous compile in which the CREATE_MASTER control option was specified. The RECOMPILE keyword must appear in the first source record to the SDL/UPL compiler.

The XMAP compiler option is not allowed when the RECOMPILE compiler option is specified.

RECOMPILE_TIMES

The keyword RECOMPILE_TIMES causes the SDL/UPL compiler to print the start and stop times of each phase of the binding pass when the CREATE_MASTER or RECOMPILE control option is specified.

SEQ

The keyword SEQ causes the file labeled NEWSOURCE to be resequenced using <base-sequence-number> as the beginning sequence number and <increment> as the incrementing value. The default is to begin at sequence number 1000 and to increment by 1000.

SGL

Refer to the SINGLE keyword.

SINGLE

The keyword SINGLE causes the program listing to be single spaced. The default is single space.

SIZE

The keyword SIZE causes the SDL/UPL compiler to print the code segment sizes by name at the end of the program listing.

SUPPRESS

The keyword SUPPRESS causes the SDL/UPL compiler to suppress warning messages in the program listing. To suppress sequence error messages, specify NO CHECK.

UNDERSCORES_IN_FILE_NAMES

The keyword UNDERSCORES_IN_FILE_NAMES is used in conjunction with the CONVERTDOTS compiler option and causes all the period (.) characters in a file identifier to be converted to the underscore (_) character.

USEDOTS

The keyword USEDOTS allows the use of the period (.) character as a separator in identifiers. The period (.) character separator remains in all output file identifiers.

VOID

The keyword VOID interacts with certain records in the file labeled SOURCE. All records that have a sequence number which is equal to the sequence number on the VOID compiler option record and up to the sequence number specified by <ending-sequence-number> are excluded in the NEWSOURCE file and in the compilation. If <ending-sequence-number> is not specified, only the record with the sequence number corresponding to the sequence number of the VOID control option is omitted. The VOID control option does not delete records in the secondary source file that is labeled CARDS.

VSSIZE

The keyword VSSIZE causes the number specified by <vssize-number> to be used as the value stack size. The SDL/UPL compiler generates the default value stack size used by each program based on standard algorithms.

vssize-number

This field can contain any number and specifies the size in bits of the value stack.

XMAP

The keyword XMAP causes the SDL/UPL compiler to generate a file for use by the SDL/XMAP program. The S-machine code generated is associated with the sequence number in the source file. The name of the file passed to the SDL/XMAP program is XMAPnnnnnn, where nnnnnn is the job number of the compile. The file attributes of the cross-map printer file can be controlled through the use of XMAP_LINE internal file identifier in the SDL/UPL compiler.

The XMAP compiler option is not allowed with a partial recompilation or a create-master compile. The partial recompilation is invoked by the RECOMPILE compiler option and the create-master compile is invoked by the CREATE_MASTER compiler option.

XREF

The keyword XREF causes the SDL/UPL compiler to generate a file for use by the SDL/XREF program in which all identifiers specified in the source file are printed in alphabetical order with the associated sequence number. The name of the file passed to the SDL/XREF program by the SDL/UPL compiler is XREFmmddyy/<time>, where mm is the month, dd is the day, yy is the year, and <time> is the current system time. The file attributes of cross-reference printer file can be controlled through the use of XREF_LINE internal file identifier in the SDL/UPL compiler. The EXPAND_DEFINES compiler directing option must be specified in order for the SDL/UPL compiler to cross reference define identifiers.

XREF_LITERALS

The keyword XREF_LITERALS causes the SDL/UPL compiler to generate a file for use by the SDL/XREF program in which all literals specified in the source file are printed in alphabetical order with the associated sequence number. The name of the file passed to the SDL/XREF program by the SDL/UPL compiler is XREFmmddyy/<time>, where mm is the month, dd is the day, yy is the year, and <time> is the current system time. The file attributes of cross-reference printer file can be controlled through the use of XREF_LINE internal file identifier in the SDL/UPL compiler. When used in conjunction with the XREF compiler option, the literals and identifiers are merged together into the same file.

XREF_ONLY

The keyword XREF_ONLY causes the SDL/UPL compiler to generate a file for use by the SDL/XREF program in which all identifiers specified in the source file are printed in alphabetical order with the associated sequence number. The name of the file passed to the SDL/XREF program by the SDL/UPL compiler is XREFmmddyy/<time>, where mm is the month, dd is the day, yy is the year, and <time> is the current system time. The file attributes of cross-reference printer file can be controlled through the use of XREF_LINE internal file identifier in the SDL/UPL compiler. The SDL/UPL compiler does not compile the program. The EXPAND_DEFINES compiler directing option must be specified in order for the SDL/UPL compiler to cross reference define identifiers.

Examples:

```
                 Positions in the Source Record
-----------------------------------------------------------------
1 |<-------------- 2-72 ------------------------------------>| 73-80

&  XREF XMAP XREF_LITERALS CHECK FORMAL_CHECK              00000100

&  LIBRARY "DEFINES"                                        00000200

&  LIST CONTROL
```

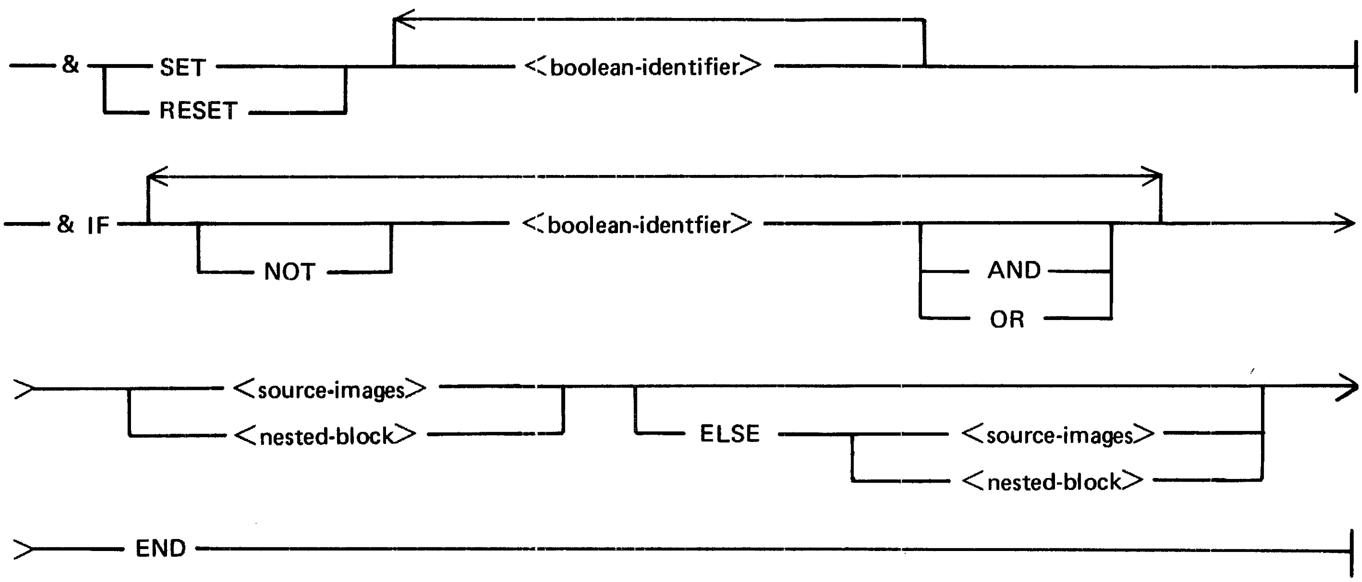
CONDITIONAL COMPILATION

The conditional compilation facility selectively includes or excludes blocks of source images without physically adding or removing the source images.

The conditionally included records are always written to a new file (if one is created), whether or not the records are compiled. However, if the conditionally excluded records are to be printed with the source listing, the LISTALL compiler option must be specified. If the LISTALL compiler option is not specified, only those conditionally included source images that are compiled are printed.

All source images containing conditional compilation statements must have an ampersand (&) character in position 1 of the record, with the exception of <nested-block>. In addition, a complete conditional inclusion statement must be contained in one ampersand record. The conditional statement can be specified in free-form format on the source record in positions 2 through 72. Positions 73 through 80 can contain sequence numbers.

SDL and UPL Syntax:



Syntax Semantics:

&

The ampersand (&) character specifies that a conditional compilation statement follows.

SET

The keyword SET causes <boolean-identifier> to have a TRUE value.

RESET

The keyword RESET causes <boolean-identifier> to have a FALSE value.

boolean-identifier

This field can be any identifier and specifies the boolean indicator used to set, reset, or to test in the IF condition compilation statement.

IF

The keyword IF designates <boolean-identifier> to be tested for a TRUE or FALSE value.

NOT

The keyword NOT negates the current value of <boolean-identifier> in the test for a TRUE or FALSE value.

AND

The keyword AND requires that the two boolean identifiers both evaluate to a TRUE value in order for the condition to be TRUE.

OR

The keyword OR requires that at least one of the boolean identifiers evaluate to a TRUE value in order for the condition to be TRUE.

source-images

This field can have any group of valid SDL/UPL statements specified and are included in the compilation of the program if the evaluation of the IF condition is TRUE.

nested-block

This field can be another IF conditional compilation statement.

ELSE

The keyword ELSE causes <source-images> or <nested-block> to be included in the compilation of the program if the evaluation of the IF condition is FALSE.

Example:

The following is the result of compiling the example.

```
               Positions in the Source Record
--------------------------------------------------------------------------------
1 |<--------------------- 2-72 ---------------------------->| 73-80

&   SET A B DEBUG                                                      00000100
&   RESET D E                                                          00000200
DECLARE (A,B) FIXED;                                                   00000300
&   IF A AND E                                                         00000400
A := B;                                                                00000500
&   ELSE                                                               00000600
A := B CAT B+5;    %WHOLE SOURCE IMAGE IS INCLUDED                     00000700
&   IF DEBUG                                                           00000800
B := A;                                                                00000900
&   END                                                                00001000
&   END                                                                00001100
&   IF B OR D                                                          00001200
BUMP B;                                                                00001300
&   ELSE                                                               00001400
BUMP A;                                                                00001500
&   END                                                                00001600
```

The following is the result of compiling the example.

```
DECLARE (A,B) FIXED;                                                   00000300
A := B CAT B+5;       %WHOLE SOURCE IMAGE IS INCLUDED                  00000700
B := A;                                                                00000900
BUMP B;                                                                00001300
```

FUNCTIONS OF EACH COMPILER PASS

The first compiler pass merges patches from the file labeled CARDS with the file labeled SOURCE, expands all definitions declared with the DEFINE statement, handles file declarations, and writes the results to an intermediate file labeled PFILE.

The second pass uses the PFILE to parse data declarations, forward procedure declarations, switch-file declarations, and procedure declarations, including formal parameter declarations. The results are written to a second intermediate file labeled IFILE.

The third pass uses the IFILE to parse statements and generate object code for all statements. If the CREATE_MASTER or RECOMPILE compiler control options are not specified, this object code is bound into a final code file.

The fourth (bind) pass is invoked only if the CREATE_MASTER or RECOMPILE is specified. In this case, the SDL compiler binds intermediate code file information into the final code file.

The organization of an SDL/UPL program is reflected in the structure of each of the three main passes of the SDL/UPL compiler. Each pass consists of: 1) a procedure that handles declarations, 2) a procedure which handles procedure declarations (this procedure handles declarations, procedures, and statements using recursion), and 3) a procedure which handles statements. At the beginning of each pass the initialization procedure is invoked and at the end of each pass the termination procedure is invoked. The last (bind) pass consists of four parts. These parts are the combine phase (if CREATE_MASTER is specified), the merge phase (if RECOMPILE is specified), the address-fixup phase, and the create-final-code-file phase.

SECTION 11
HOW TO WRITE AN SDL/UPL PROGRAM

GENERAL

The writing of a computer program presupposes an understanding of the problem to be solved and a selection of the programming language most suitable to efficiently solving that problem. Assuming that these conditions are satisfied, the following considerations should be kept in mind as a guide in writing a SDL/UPL source language program.

WRITING RULES

The SDL/UPL compiler accepts a card-image input file of records where columns 1 through 72 can be used for statements, declarations, or comments and where columns 73-80 are the record sequence numbers and/or identification field.

The coding can be specified in a completely free form, that is, any number of statements, declarations, or comments can appear on a single record or over as many records as desired. Column 72 is considered adjacent to column 1 of the next record. Extra spaces can be used freely throughout the SDL/UPL record line to improve the readability of the text. A percent sign (%) character denotes that the rest of the record is composed of comments. It can be used to delimit the scan procedure, thus increasing compile speed. The following shows an example of using the IF statement.

```
IF X EQL Y THEN X := 0; % Each line on the page represents
           ELSE X := 1; % a separate record.
```

FORM OF AN SDL/UPL PROGRAM

Programs are divided into logical units called procedures, each having a procedure head at its beginning and being terminated with and END statement. Procedures have an internal structure as described in Section 7. A procedure has a definite ordered relationship to all other procedures within a program from either a side-by-side (parallel procedure) or subordinate (nested procedure) position in that program. The ordering inherently defines the scope or range of an identifier and the procedures that can invoke from a given procedure.

The main program (lexicographic level 0) is considered a procedure except that it has no procedure head or END statements and therefore cannot be recursively invoked.

Identifiers and nested procedures that are used within a procedure must be declared and completed before any executable statements in that procedure.

The outer-most procedure is considered to be the program. The procedures contained within the program are considered nested at least one level down, that is, they are on lexicographic level 1 or greater, with the maximum depth of 15 sublevels for UPL and 31 sublevels for SDL. Refer to Section 3 for a description on the structure of an SDL/UPL program.

Execution of an object SDL/UPL program starts at the first executable statement in the outermost procedure and is the statement that immediately follows all nested procedures. The statements are performed successively from statement to statement within the outermost procedure or until a STOP statement is encountered.

Since the record line format in an SDL/UPL program is very flexible, it is suggested that statement levels be indented on new records to improved the documentation references and the general understanding of a program. Thus, each new procedure can be indented to a new margin, and its corresponding END statement can be placed on that same margin. Also, since statements can contain other statements (such as DO, IF, and CASE), each lower level statement level can be indented. When a higher level is resumed, its statements should be placed at the proper level margin. This is only a suggestion. Indentation of statements does not affect the operation of the SDL/UPL program.

Studying the examples and the detailed descriptions of the SDL/UPL statements and declarations in this manual should aid in understanding SDL/UPL program structure.

CODING EXAMPLES

Two SDL/UPL programs that read a record, extract 11 fields of seven columns each, convert each field to a FIXED number are shown in Figures 11-1 and 11-2. Each shows one method that can be used to perform this task. Figure 11-1 shows a straight-forward approach and Figure 11-2 shows the recursive-procedure technique which is more typical of an SDL/UPL program.

```
?COMPILE TEST WITH UPL;
?DATA CARDS
DECLARE BUFFER CHARACTER (80),
        CHAR   CHARACTER (24),
        (F, M, COL) FIXED;
FILE IN  (DEVICE = DISK,
          RECORDS = 80/1),
     OUT (DEVICE = PRINTER,
          RECORDS = 132/1);
OPEN IN INPUT;
OPEN OUT OUTPUT NEW;
COL := -7;
READ IN (BUFFER);
DO EXTRACT_FIELD FOREVER;
   IF (BUMP COL BY 7) GTR 70 THEN UNDO EXTRACT_FIELD;
   F := CONV (SUBSTR (BUFFER, COL, 7), FIXED);
   M := 0;
   DO CONVERT_BITS FOREVER;
      SUBSTR (CHAR, M, 1) := CONV (SUBBIT (F, M, 1), CHARACTER);
      IF (BUMP M) GTR 23 THEN UNDO CONVERT_BITS;
   END CONVERT_BITS;
   WRITE OUT (CHAR);
END EXTRACT_FIELD;
CLOSE IN;
CLOSE OUT;
STOP;
FINI;
?END
```

Figure 11-1. Straight Forward SDL/UPL Program

```
?COMPILE TEST WITH UPL;
?DATA CARDS
$ CSSIZE 40
$ PPSSIZE 50
FILE IN  (DEVICE = DISK,
          RECORDS = 80/1),
     OUT (DEVICE = PRINTER,
          RECORDS = 132/1);
DECLARE BUFFER CHARACTER (80),
        CHAR   CHARACTER (24);
PROCEDURE PROCESS_FIELD (W, Y);
FORMAL (W, Y) FIXED;
  SUBSTR (CHAR, (Y-1), 1) := CONVERT (SUBBIT(W, (Y-1), 1), CHARACTER);
  IF (DECREMENT Y) GTR 0 THEN PROCESS_FIELD (W, Y); % Recursive Call
                         ELSE WRITE OUT (CHAR);
  RETURN;
END PROCESS_FIELD;
PROCEDURE PROCESS_BUFFER (X);
FORMAL (X) FIXED;
  DECLARE COL FIXED,
          F   FIXED;
  COL :=    7 * -(X - 11);
  F := CONVERT (SUBSTR (BUFFER, COL, 7), FIXED);
  PROCESS_FIELD (F, 24);
  IF (DECREMENT X) GTR 0 THEN PROCESS_BUFFER (X);  % Recursive Call
                         ELSE RETURN;
END PROCESS_BUFFER;
OPEN IN INPUT;
OPEN OUT OUTPUT;
READ IN (BUFFER);
PROCESS_BUFFER (11);
STOP;
FINI;
?END
```

Figure 11-2. SDL/UPL Program Using Recursive-Procedure Technique

APPENDIX A
RESERVED AND SPECIAL WORDS

The reserved words used by the SDL/UPL compiler are listed below.

ACCEPT	FILE	REDUCE
AND	FILLER	REFER
AS	FINI	REFERENCE
BASE	FIXED	REMAPS
BIT	FORMAL	RETURN
BUMP	FORMAL_VALUE	RETURN_AND_ENABLE_INTERRUPTS
BY	FORWARD	SEARCH_DIRECTORY
CASE	FROM	SEEK
CAT	GEQ	SEGMENT
CHANGE	GTR	SEGMENT_PAGE
CHARACTER	IF	SKIP
CLEAR	INTRINSIC	SPACE
CLOSE	LEQ	STOP
DECLARE	LOCK	SUBBIT
DECREMENT	LSS	SUBSTR
DEFINE	MOD	SWITCH_FILE
DISPLAY	NEQ	THEN
DO	NOT	TO
DUMMY	OF	UNDO
DYNAMIC	ON	USE
ELSE	OPEN	VARYING
END	PAGED	WRITE
EQL	PROCEDURE	WRITE_FILE_HEADER
ENTER_COROUTINE	READ	ZIP
EXIT_COROUTINE	READ_FILE_HEADER	
EXOR	RECORD	

The special words used by the SDL/UPL compiler are listed below.

ACCESS_FILE_INFORMATION	M_MEM_SIZE
BASE_REGISTER	MONITOR_CHANGE
BINARY	MONITOR_RESET
BINARY_SEARCH	MONITOR_SET
CHANGE_STACK_SIZES	NAME_OF_DAY
CHARACTER_FILL	NAME_STACK_TOP
CHAR_TABLE	NDL_OP
COMMUNICATE	NEXT_ITEM
COMPILE_CARD_INFO	NEXT_TOKEN
COMMUNICATE_WITH_GISMO	NOTRACE
CONTROL_STACK_BITS	NULL
CONTROL_STACK_TOP	OVERLAY
CONSOLE_SWITCHES	PARITY_ADDRESS
CONV	PREVIOUS_ITEM
CONVERT	PROGRAM_SWITCHES
DATA_ADDRESS	READ_CASSETTE
DATA_LENGTH	READ_FPB

DATA_TYPE
DATE
DC_INITIATE_IO
DEBLANK
DECIMAL
DELIMITED_TOKEN
DESCRIPTOR
DISABLE_INTERRUPTS
DISPATCH
DISPLAY_BASE
DMS_CALL
DUMP
DUMP_FOR_ANALYSIS
DYNAMIC_MEMORY_BASE
ENABLE_INTERRUPTS
ERROR_COMMUNICATE
EVALUATION_STACK_TOP
EXECUTE
FETCH
FETCH_COMMUNICATE_MSG_PTR
FETCH_AND_SAVE
FIND_DUPLICATE_CHARACTERS
FREEZE_PROGRAM
GROW
HALT
HARDWARE_MONITOR
HASH_CODE
HASH_UNPACK
INITIALIZE_VECTOR
INTERROGATE_INTERRUPT_STATUS
LENGTH
LIMIT_REGISTER
LOCATION
MAKE_DESCRIPTOR
MAKE_READ_ONLY
MAKE_READ_WRITE
MESSAGE_COUNT
READ_OVERLAY
REFER_ADDRESS
REFER_LENGTH
REFER_TYPE
REINSTATE
RESTORE
REVERSE_STORE
SAVE
SAVE_STATE
SEARCH_LINKED_LIST
SEARCH_SERIAL_LIST
S_MEM_SIZE
SEARCH_SDL_STACKS
SORT
SORT_DELETE
SORT_FILE_FIXUP
SORT_MERGE
SORT_RETURN
SORT_SEARCH
SORT_STEP_DOWN
SORT_SWAP
SORT_UNBLOCK
SWAP
SPO_INPUT_PRESENT
THAW_PROGRAM
THREAD_VECTOR
TIME
TRACE
TRANSLATE
VALUE_DESCRIPTOR
WAIT
WRITE_FPB
WRITE_OVERLAY
X_ADD
X_DIV
X_MOD
X_MUL
X_SUB

APPENDIX B
THE SDL S-MACHINE

The SDL S-machine is described in this appendix.

COMPONENTS OF THE SDL S-MACHINE

The five basic components of the SDL S-Machine are described as follows.

Base-Limit Area
Run Structure Nucleus
Code segments and segment dictionaries
File Information Blocks (FIB) and FIB dictionary
Registers

Base-Limit Area

The base-limit area is the memory area for program data. The contents of this area are directly addressable and modifiable by SDL S-operators. This area is bound by the base and limit registers. All data addresses in the S-machine are expressed as a bit offset from the base register. Addresses are made machine absolute by adding the contents of the base register to the address. The area is broken into two divisions: 1) static memory (from base register to dynamic memory base) which is occupied by the SDL stacks, and 2) dynamic memory (from dynamic memory base to the limit register) which is used for virtual data memory. SDL paged-array page tables and resident pages can occupy the dynamic memory area.

Run Structure Nucleus

The Run Structure Nucleus of a program contains information used by the MCP and the SDL interpreter to run an SDL/UPL program.

Code Segment and Segment Dictionaries

Code segments are virtual as in the other S-machines, but the Code Segment Dictionary is segmented, corresponding to the page-segment concept in the SDL/UPL language. Each entry in a Master Segment Dictionary represents a page of segments in the source program and points to a subdictionary with the entries for those segments. The Master Segment Dictionary cannot be overlayed; the subdictionaries can be overlaid.

File Information Block and FIB Dictionary

The File Information Block (FIB) and the FIB Dictionary appear in memory, one FIB for each open file in use by the running program. The FIB and FIB Dictionary are used by the B 1000 operating system for input/output operations.

Registers

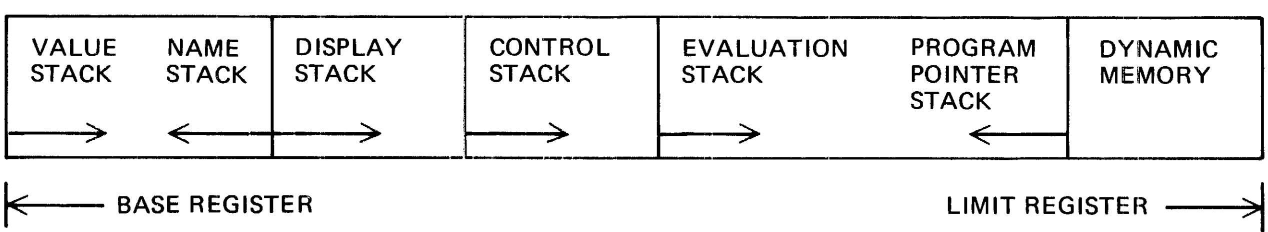
Registers can be hardware registers or they can be stored in the Run Structure Nucleus, depending on the state of the S-machine. The exact format and number of registers is important only to the SDL Interpreter. Logically, the registers consist of the next instruction pointer (page, segment and displacement), the current lexic level, and the stack top pointers for all stacks. The current lexic level and Display stack pointer are contained in the same register. The registers contain enough information about the stacks to check for stack overflows. Underflows are not detected. Registers are initialized from the scratchpad area of the Program Parameter Block in the code file. The format of an SDL/UPL scratch pad in the code file follows.

```
01 SCRATCHPAD,
   02 FILLER            BIT(48),
   02 PPS_BASE          BIT(24),

   02 ES_BASE           BIT(24),
   02 ES_PPS_BITS       BIT(24),
   02 VS_BASE           BIT(24),
   02 FILLER            BIT(24),
   02 CS_BASE           BIT(24),
   02 CS_BITS           BIT(24),
   02 NS_BASE           BIT(24),
   02 FILLER            BIT(24),
   02 DISPLAY_BASE      BIT(24),
   02 FILLER            BIT(4),
   02 PROFILE_FLAG      BIT(1),
   02 FILLER            BIT(19),
   02 VS_BITS           BIT(24),
   02 NS_BITS           BIT(24),
   02 ES_BITS           BIT(24),
   02 PPS_BITS          BIT(24);
```

The concepts just presented are common to all the B 1000 S-machines. The SDL/UPL language does not use data segments and data segment dictionaries. These are handled by way of an SDL intrinsic.

THE BASE-LIMIT AREA

The base-limit area of main memory is divided as shown in Figure B-1. The arrows indicate the directions of growth.



G18311

Figure B-1. Base-Limit Area of an SDL/UPL Program

Value Stack

The Value stack contains the value of a data item. The length and data type are kept in the Name and Evaluation stack.

Name Stack

The Name stack contains the data descriptors, each 48 bits long with one descriptor for every data identifier which is currently active (not necessarily addressable) in the program. The data descriptor for an array is 96 bits long and occupies two Name stack entries. The Name stack is divided into stack frames, each frame containing the descriptors for the names declared in one invocation of a procedure. Not all of these stack frames contain currently accessible descriptors.

Display Stack

The Display stack contains pointers into the Name stack, one pointer for each lexic level less than or equal to the current lexic level. Each pointer locates the base of the frame for currently addressable names at that lexic level. Each pointer entry is 32 bits long.

Control Stack

The Control stack contains the Name stack pointers which locate the stack frames for every active procedure. Each time a procedure which requires local data or parameter allocations, that is, requires space on the Name stack, is entered, a new entry is pushed onto the Control stack to point to its Name stack frame.

Because the data associated with Name stack descriptors is contained in the Value stack, this stack is also divided into frames and the base of each frame is recorded in the Control stack as it is allocated. In addition to these two pointers, each entry contains the lexic level of the calling procedure and the lexic level of the current entry. These are used by the S-machine to maintain the Display stack. Figure B-2 shows the format of a Control stack entry.

NAME STACK POINTER	EXITED LEXIC LEVEL	ENTERED LEXIC LEVEL	VALUE STACK POINTER
20 BITS	4 BITS	4 BITS	20 BITS

G18312

Figure B-2. Format of Control Stack Entry

Evaluation Stack

The Evaluation stack is used to hold data descriptors for the evaluation of expressions, which are compiled into reverse polish strings. The Evaluation stack is also used to build actual parameter descriptors before they are transferred to the Name stack for a procedure call. Space for data during expression evaluation is allocated on top of the Value stack, which is kept up to date as descriptors are pushed onto or removed from the Evaluation stack.

Program Pointer Stack

The Program Pointer stack contains the next instruction pointer of a program. With the exception of the cycle operator used for looping, all transfers of control in the SDL S-machine are done by means of call-type operators. The next instruction pointer is saved for subsequent return by pushing it onto the Program Pointer stack. Figure B-3 shows the format of an entry in the Program Pointer stack.

SEGMENT	PAGE	DISPLACEMENT
6 BITS	6 BITS	20 BITS

G18313

Figure B-3. Format of the Program Pointer Stack

DATA DESCRIPTOR

The data descriptor is the descriptor in the Name and Evaluation stack.

Figure B-4 shows the format for a 48-bit long simple, scalar descriptor.

TYPE	LENGTH	ADDRESS
8 BITS	16 BITS	24 BITS

G18314

Figure B-4. Format for a 48-bit Long Simple Descriptor

The address, expressed in bits, is specified as a bit offset from the base register regardless of the data type.

One of the bits in the type field indicates whether a descriptor is an array descriptor. When this bit is on, an additional 48 bits of information is appended. Figure B-5 shows the format for an array descriptor.

TYPE	LENGTH OF ENTRY	ADDRESS OF ARRAY
PAGE SUBSCRIPT SIZE	LENGTH BETWEEN ENTRIES	NUMBER OF ENTRIES
8 BITS	16 BITS	24 BITS

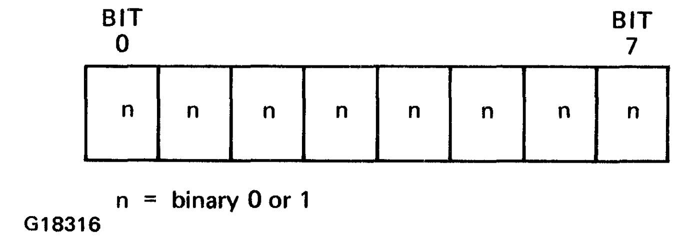
G18315

Figure B-5. Format of an Array Descriptor

The page subscript size is used only when the paged array bit is ON in the type field. The page subscript size specifies the number of bits to shift an array subscript to obtain the corresponding page subscript. Page subscript sizes are always a power of two.

The length between entries is the difference between the address of one element and the address of the previous element. This length must be greater than or equal to the length of one entry.

The type field of a descriptor has a single format, even though some bits are not meaningful in all contexts. Figure B-6 shows the bit format of the type field.



Bit Number	Binary Value	Description
0	0	Indicates a Name stack entry.
	1	Indicates a Value stack entry. This bit is only used when the descriptor is on the Evaluation stack.
1	0	Indicates a self-relative descriptor.
	1	Indicates a non-self-relative descriptor. This bit must be ON if bit 2 equals 1.
2	0	Descriptor is not an array descriptor.
	1	Descriptor is an array descriptor.
3	0	Not contiguous array.
	1	Contiguous array. The length between elements equals the length of one element (bit 2 must be equal to 1).
4-5	00	Indicates a BIT data type.
	01	Indicates a FIXED data type.
	10	Indicates a CHARACTER data type.
	11	Indicates a VARYING data type. Used only in the type field of inline descriptors which are arguments of a construct descriptor formal check (CDFC) operator and in the argument to a return formal check (RTNC) operator. The CDFC operator also uses bit 6 in a different way: it indicates a varying array bound. Inline descriptors for other operators use bit 0 to indicate the presence of a filler field. Refer to INLINE DESCRIPTOR FORMATS in this appendix.
6	0	Not a paged array.
	1	Indicates a paged array (bit 2 must also equal 1).
7	0	Not a VARYING length.
	1	Indicates a VARYING length. Used only in the type field of inline descriptors which are arguments of a construct descriptor formal check (CDFC) operator and in the argument to a return formal check (RTNC) operator. The CDFC operator also uses bit 6 in a different way: it indicates a varying array bound. Inline descriptors for other operators use bit 0 to indicate the presence of a filler field. Refer to INLINE DESCRIPTOR FORMATS in this appendix.

Figure B-6. Format of the Type Field

When the data item is 24 bits or less in length, it can be contained directly in the address portion of the descriptor, thus requiring less storage space. In this case, the descriptor is said to be self-relative and the non-self-relative bit is off.

The use of the name-value bit, in the Evaluation stack, is to distinguish between descriptors that had an associated value loaded on the Value stack when they were pushed on the Evaluation stack, and those that did not. The purpose is to signal that a data item must be removed from the Value stack whenever this descriptor is removed from the Evaluation stack. The bit can be set only in non-self-relative descriptors.

PAGED ARRAY DESCRIPTORS

When the paged array bit is ON in an array descriptor, the address field of the descriptor does not point directly to the array, but is initialized to 0 (zero). An array load operator (ALA, AL) detects the first access to the array and invokes the SDL virtual memory manager to build a page table in dynamic memory. This table is non-overlayable and the descriptor address field is set to the page table address. Figure B-7 shows the format of a paged array descriptor.

STATUS	ADDRESS
4 BITS	24 BITS

G18317

Bit Number	Bit Value	Description
0	0	Address is present and is a disk address.
	1	Address is present and is a base relative memory address.
1	0	Not to be read.
	1	To be read. The next time this page is rolled out, this bit is set to 0 and bit 2 is set to 1.
2	0	This paged array cannot be overlaid without rolling it out to disk.
	1	This page array can be overlaid without rolling it out to disk.
3		This bit is not used.

Figure B-7. Format of a Paged Array Descriptor

An address field of 0 (zero) indicates that this is a previously unaccessed paged array and can be created without rolling it in.

ACCESS OF DATA ADDRESSES

Data addresses are accessed with the SDL S-machine language by means of descriptors on the Name stack. At any point in an SDL/UPL program, every accessible data item can be described by the lexic level at which it was declared by its ordinal location (occurrence number) within the declaration section at that level. A data address consists of these two numbers which uniquely locate a descriptor in the Name stack. Addressing is done by using the Display stack to locate the Name stack frame corresponding to the required lexic level, and by using the occurrence number to locate the descriptor within that frame. To make data addresses more compact, they have a type field which indicates the sizes of the other fields. Figure B-8 shows the format of a data address.

TYPE	LEXIC LEVEL	OCCURRENCE NUMBER
2 BITS	1 OR 4 BITS	5 OR 10 BITS

G18318

Type	Lexic Level Bits	Occurrence Number Bits	Total Bits
00	4	10	16
01	4	5	11
10	1	10	13
11	1	5	8

When only one bit is used for lexic level, 0 indicates lexicographic level 0 and 1 indicates the current lexic level.

Figure B-8. Format of a Data Address

CODE ADDRESSES

Code addresses appear as arguments of operators which affect transfers of control. They are divided into three parts: the page number which selects the segment dictionary page, the segment which selects the segment dictionary entry within that page, and the displacement which specifies a bit offset within the segment. To make data code addresses more compact, these numbers are encoded in different field sizes which are determined by a type field. Figure B-9 shows the format of code addresses.

TYPE	SEGMENT	PAGE	DISPLACEMENT
3 BITS	0 or 6 BITS	0, 4, or 6 BITS	12, 16, or 20 BITS

G18319

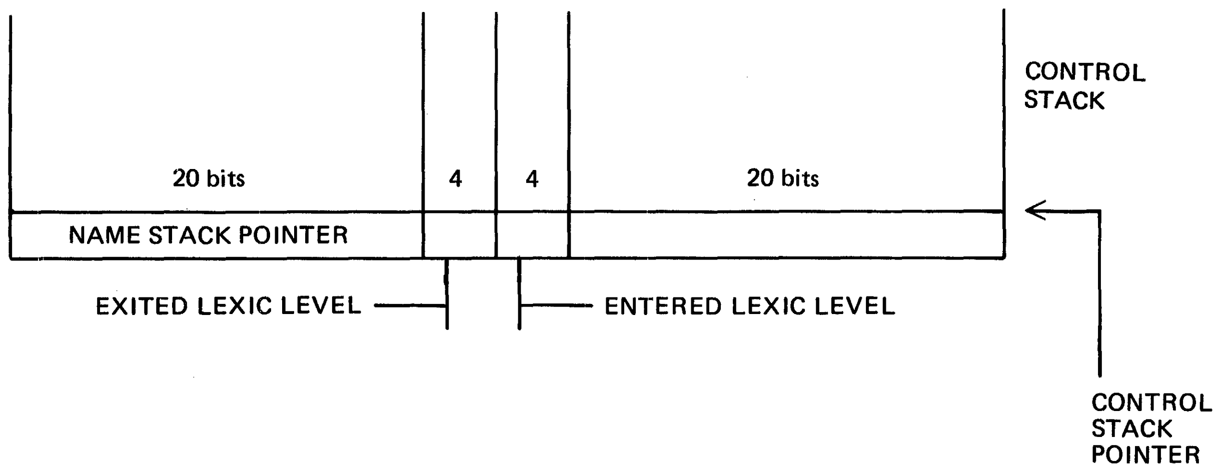
Type	Segment Bits	Page Bits	Displacement Bits	Total Bits
000	Current	Current	12	15
001	Current	Current	16	19
010	6	Current	12	21
011	6	Current	16	25
100	6	4	12	25
101	6	4	16	29
110	6	4	20	33
111(0)	Null Address			
(1)	6	6	20	36

Type 111 with the following bit OFF and with a Null Address, is used only to mark null entries in a CASE operator. The length of this code type is the same as the code address type specified by the CASE operator.

Figure B-9. Format of Code Addresses

FORMAT OF THE CONTROL STACK AND SCRATCH PAD

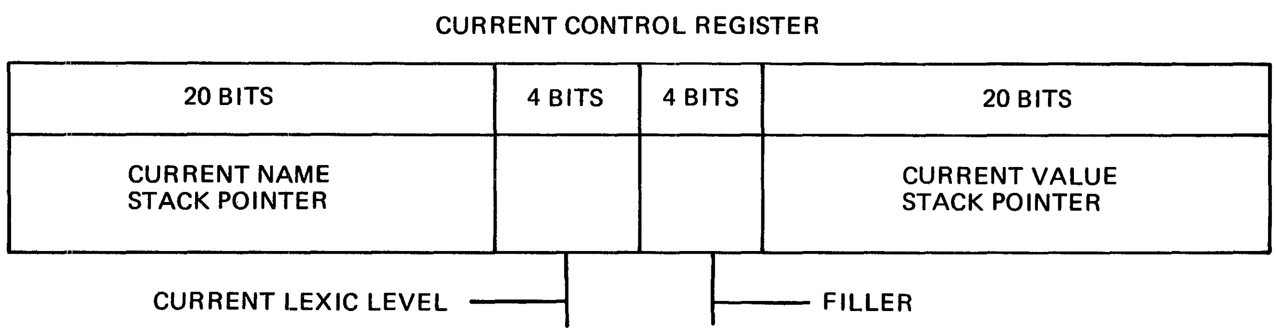
Figure B-10 shows the format of the CONTROL stack.



G18320

Figure B-10. Format of the Control Stack

Figure B-11 shows the current Control stack information contained in the scratch pad.



G18321

Figure B-11. Format of Control Stack Information in Scratch Pad

The following SDL declaration shows the format of the Control stack and Current Control register.

```
DECLARE
  01 CONTROL_STACK (CS_SIZE)              BIT(48),
     02  CS_NSP                           BIT(20),
     02  CS_EXITED_LL                     BIT(4),
     02  CS_ENTERED_LL                    BIT(4),
     02  CS_VSP                           BIT(20),

  01 CURRENT_CONTROL                      BIT(48),
     02  CURRENT_NSP                      BIT(20),
     02  CURRENT_LL                       BIT(4),
     02  FILLER                           BIT(4),
     02  CURRENT_VSP                      BIT(20),
 (CSP,TCSP)                               FIXED;

  CSP:=0;
```

The following SDL operators are used in the Control stack mechanism.

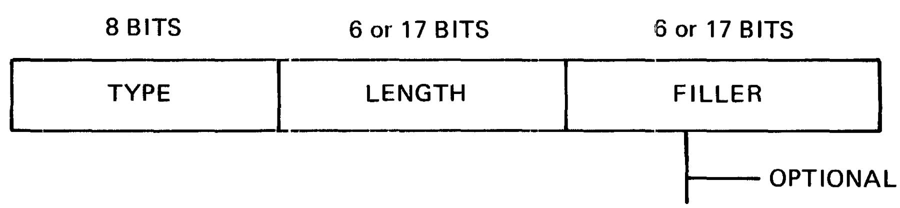
SDL Operator	Descriptor
MKS	MARK STACK
CDFM	CONSTRUCT DESCRIPTOR, FORMAL
CDFC	CONSTRUCT DESCRIPTOR, FORMAL CHECK
MKU	MARK STACK AND UPDATE
EXIT	EXIT
RTRN	RETURN
RTNC	RETURN FORMAL CHECK
XTEI	EXIT, ENABLE INTERRUPTS

INLINE DESCRIPTOR FORMATS

Inline descriptors, which are used by the construct descriptor operators, have the following format. The type field has the same format as that in the data descriptors.

Simple Data Descriptor Format

Figure B-12 shows the format of a simple data descriptor.



G18322

NOTES

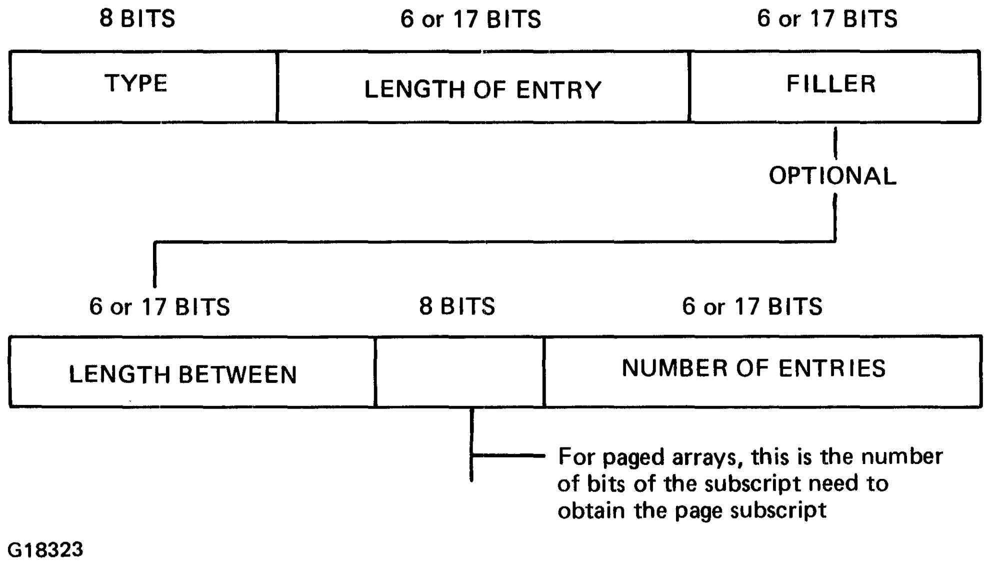
The filler option is present only when bit 0 of the type field is ON.

Bit 2 of the type field is always 0 (zero).

Figure B-12. Format of a Simple Data Descriptor

Array Descriptor Format

Figure B-13 shows the format of an array descriptor.



NOTES

The filler option is present only when bit 0 of the type field is equal to 1.

The length between option is present only when bit 3 of the type field is equal to 0 (zero).

Bit 2 of the type field is always equal to 1.

The page subscript size field is present only when bit 6 of the type field is equal to 1.

If bit 6 of the type field is on, then bits 0 (zero) and 3 are equal to 0 (zero).

The field that contains six bits always has a 0 (zero) in the leftmost bit position. The field that contains 17 bits always has a 1 in the leftmost bit position.

First Bit	Meaning
0	5 bits follow
1	16 bits follow

Figure B-13. Format of an Array Descriptor

USE OF THE EVALUATION STACK

Many of the SDL/UPL S-machine operators (S-ops) take operands from or leave results on the Evaluation stack. Only the descriptor of the operand is on the Evaluation stack while the data (the value of the operand) can be in the descriptor or elsewhere in the base-limit area. Conceptually, the S-operator is working with an operand. There are two classes of operands or results on the Evaluation stack: address operands and value operands.

Address Operand

The address operand is a pointer to the value of a declared data item. The descriptor on the Evaluation stack is non-self-relative and its name-value bit is off. This type of operand is appropriate for use as the destination of an S-operator that moves data.

A particular S-operator often requires that its operands be of a particular class. It does not make sense, for example, for the destination operand of a STOD (store destructive) to be a value operand. Some S-operators put other restrictions on their operands, usually concerning type or length. Unless specifically indicated, these restrictions are not checked by the interpreter and, if not met, the results of the operations are undefined.

Value Operands

There are two classes of value operands. These are self-relative operands and non-self-relative operands.

Self-Relative

The descriptor on the Evaluation stack is marked self-relative and its name-value bit is equal to 0. Instead of the address field of the descriptor being a pointer to the data, the data itself is contained in the address field of the descriptor.

Non-Self-Relative

The descriptor on the Evaluation stack is marked non-self-relative and its name-value bit is equal to 1. The data is on top of the Value stack, located by the address field in the descriptor. When this type of operand is removed from the Evaluation stack, its value also is removed from the Value stack.

Value operands are temporary values as opposed to actual variables of the program.

INSTRUCTION SET

The instruction set in the SDL S-machine language contains operation codes that are four, six, ten, or thirteen bits in length. The lengths have been assigned according to static frequency of the S-operator, thus compacting code space as much as possible.

Relational Operators

The following are the relational operators.

Name	Mnemonic	Operation Code
EQUAL TO	EQL	1010 01
LESS THAN	LSS	1111 01 1010
LESS THAN OR EQUAL TO	LEQ	1111 00 1110
GREATER THAN	GTR	1111 00 1001
GREATER THAN OR EQUAL TO	GEQ	1111 00 1101
NOT EQUAL TO	NEQ	1010 10

Arithmetic Operators

The following are the arithmetic operators.

Name	Mnemonic	Operation Code
ADD	ADD	1011 01
SUBTRACT	SUB	1011 10
MULTIPLY	MUL	1111 00 0101
DIVIDE	DIV	1111 00 0110
MODULO	MOD	1111 00 0111
REVERSE SUBTRACT	RSUB	1111 10 1100
REVERSE DIVIDE	RDIV	1111 10 1101
REVERSE MODULO	RMCD	1111 10 1110
NEGATE	NEG	1111 01 0111
CONVERT TO DECIMAL	DEC	1111 10 1000
CONVERT TO BINARY	BIN	1111 10 1001

Extended Arithmetic Operators

The following are the extended arithmetic operators.

Name	Mnemonic	Operation Code
EXTENDED ADD	XADD	1111 11 1100 01
EXTENDED SUBTRACT	XSUB	1111 11 1100 100
EXTENDED MULTIPLY	XMUL	1111 11 1100 101
EXTENDED DIVIDE	XDIV	1111 11 1100 110
EXTENDED MODULO	XMOD	1111 11 1100 111

Logical Operators

The following are the logical operators.

Name	Mnemonic	Operation Code
AND	AND	1111 00 0001
OR	OR	1111 00 0000
EXCLUSIVE-OR	EXOR	1111 00 0010
NOT	NOT	1111 00 1011

String Operators

The following are the string operators.

Name	Mnemonic	Operation Code	Arguments
CONCATENATE	CAT	1100 11	
SUBSTRING ONE	SS1	1111 11 0100	T,V,Q,L
SUBSTRING TWO	SS2	1111 00 1000	T,V
SUBSTRING THREE	SS3	1010 00	T,V

Store Operators

The following are the store operators.

Name	Mnemonic	Operation Code
STORE DESTRUCTIVE	STOD	0010
STORE NON-DESTRUCTIVE LEFT	SNDL	1010 11
STORE NON-DESTRUCTIVE RIGHT	SNDR	1111 00 0100

Construct Descriptor Operators

The following are the construct descriptor operators.

Name	Mnemonic	Operation Code	Arguments
CONSTRUCT DESCRIPTOR BASE ZERO	CDBZ	1111 10 0100	DESCRIPTOR
CONSTRUCT DESCRIPTOR LOCAL DATA	CDLD	1110 00	N,DES#1,..., DES#n
CONSTRUCT DESCRIPTOR FORMAL	CDFM	1111 01 0001	LL,E
CONSTRUCT DESCRIPTOR FORMAL CHECK	CDFC	1111 11 1101 000	LL,E,DES#1,... DES#n
CONSTRUCT DESCRIPTOR FROM PREVIOUS	CDPR	1110 10	N,DES#1,..., DES#n
CONSTRUCT DESCRIPTOR FROM PREVIOUS AND ADD	CDAD	1110 01	N,DES#1,..., DES#n DES#n
CONSTRUCT DESCRIPTOR FROM PREVIOUS AND MULTIPLY	CDMP	1111 10 0101	N,DES#1,..., DES#n
CONSTRUCT DESCRIPTOR LEXIC LEVEL	CDLL	1111 10 0011	TYPE-LL-OC, DESCRIPTOR
CONSTRUCT DESCRIPTOR REMAPS	CDRM	1111 00 1111	DESCRIPTOR
CONSTRUCT DESCRIPTOR DYNAMIC	CDDY	1111 11 1110 000	TYPE

Load Operators

The following are the load operators.

Name	Mnemonic	Operation Code	Arguments
MAKE DESCRIPTOR	MDSC	1111 10 1010	
VALUE DESCRIPTOR	VDSC	1111 01 1000	
DESCRIPTOR	DESC	1100 10	TYPE-LL-OC
NEXT OR PREVIOUS ITEM	NPIT	1111 01 1101	V,TYPE-LL
LOAD	L	1101 00	TYPE-LL-OC
LOAD ADDRESS	LA	0000	TYPE-LL-OC
LOAD ARRAY FIELD ADDR.	LAFA	1111 11 1111 011	TYPE,LENGTH
LOAD FIELD ADDRESS	LFA	1111 11 1111 001	TYPE,OFFSET LENGTH, TYPE-LL-OC
LOAD FIELD ADDRESS FROM PREVIOUS	LFAP	1111 11 1111 010	TYPE,OFFSET LENGTH
ARRAY LOAD VALUE	AL	1111 01 1100	TYPE-LL-OC
ARRAY LOAD ADDRESS	ALA	1101 01	TYPE-LL-OC
INDEXED LOAD VALUE	IL	1111 01 0000	TYPE-LL-OC
INDEXED LOAD ADDRESS	ILA	0001	TYPE-LL-OC
INDEXED LOAD FIELD ADDRESS	ILFA	1111 11 1111 000	TYPE,OFFSET, LENGTH
LOAD LITERAL	LIT	0100	TYPE,LENGTH, LITERAL
LOAD NUMERIC LITERAL	LITN	0011	LITERAL
LOAD NUMERIC ZERO	ZOT	0101	
LOAD NUMERIC ONE	ONE	0110	
REFER	REDR	1111 11 1111 100	

Stack Operators

The following are the stack operators.

Name	Mnemonic	Operation Code
BUMP VALUE STACK POINTER	BVSP	1111 10 1011
DUPLICATE	DUP	1100 00
DELETE	DEL	1111 00 0011
EXCHANGE	XCH	1011 00
FORCE VALUE STACK	FVS	1100 01

Procedure Operators

The following are the procedure operators.

Name	Mnemonic	Operation Code	Arguments
CALL	CALL	0111	TYPE-SEG-PAGE-DISP
IF THEN	IFTH	1001	TYPE-SEG-PAGE-DISP
IF THEN ELSE	IFEL	1101 10	ADDR TYPE,TYPE-SEG-PAGE-DISP
CASE	CASE	1111 01 0100	# OF ADDR, ADD TYPE,TYPE-SEG-PAGE-DISP,..., TYPE-SEG-PAGE-DISP
UNDO	UNDO	1000	# OF LEVELS
UNDO CONDITIONALLY	UNDO	1111 01 0011	# OF LEVELS
RETURN	RTRN	1111 01 0101	# OF LEVELS
RETURN FORMAL CHECK	RTNC	1111 11 1101 001	# OF LEVELS, TYPE,LENGTH
EXIT	EXIT	1101 11	# OF LEVELS
CYCLE	CYCL	1110 11	DISPLACEMENT
MARK STACK	MKS	1011 11	
MARK STACK AND UPDATE	MKU	1111 01 1111	LL
ENABLE-DISABLE INTERRUPTS	EDI	1111 11 0101	V
EXIT-ENABLE INTERRUPTS	XTEI	1111 11 0110	V,# OF LEVELS
CO-ROUTINE ENTRY	CNTR	1111 11 1010 000	
CO-ROUTINE EXIT	CXIT	1111 11 1010 001	# OF LEVELS
DMS-CALL	DMCL	1111 11 1110 011	

Search and Scan Operators

The following are the search and scan operators.

Name	Mnemonic	Operation Code	Arguments
REDUCE	RDUC	1111 11 1001 101	VARIANTS, TYPE-LL-OC
SEARCH SDL STACKS	SSS	1111 11 1110 001	
SEARCH LINKED LIST	SLL	1111 01 1010	COMPARE TYPE
SEARCH SERIAL LIST	SSL	1111 11 1000 000	COMPARE TYPE
SORT SEARCH	SSCH	1111 11 1011 100	
THREAD VECTOR	TVEC	1111 11 1011 011	
INITIALIZE VECTOR	IVEC	1111 11 1011 000	
START STEP DOWN	SSD	1111 11 1011 010	
START SWAP	SSWP	1111 11 1011 101	
START UNBLOCK	UBLK	1111 11 1011 011	
DELIMITED TOKEN	DTKN	1111 11 1001 001	TYPE-LL-OC, DEL1,DEL2
NEXT-TOKEN	NTKN	1111 11 1001 000	TYPE-LL-OC SEPARATOR,V
DEBLANK	DBLK	1111 11 1001 010	TYPE-LL-OC
CHARACTER FILL	CHFL	1111 11 1001 100	
TRANSLATE	XLAT	1111 11 1110 101	
FIND DUPLICATE CHARACTERS	FDUP	1111 11 1001 011	

Miscellaneous Operators

The following are the other operators.

Name	Mnemonic	Operation Code	Arguments
TRANSFER MESSAGE	XFRM	1111 11 1010 010	DEST,VARIABLES SOURCE VARIABLES
HASH CODE	HASH	1111 11 1000 001	
HASP UNPACK	HASP	1111 11 1111 101	LL,ON
SWAP	SWAP	111 01 0110	
FETCH	FECH	111 00 1100	
DISPATCH	DISP	1111 01 1011	
HALT	HALT	1111 11 0010	
READ CASSETTE	RDCS	1111 01 0010	
LENGTH	LENG	1111 10 0000	

Name	Mnemonic	Operation Code	Arguments
LOAD SPECIAL	LSP	1111 01 1110	VARIANT
NDL SOPS	NDL	1111 11 1111 110	TYPE,# DESC
CLEAR ARRAY	CLR	1111 10 0111	
COMMUNICATE	COMM	1111 10 0110	
REINSTATE	REIN	1111 10 0001	
FETCH CMP	FCMP	1111 10 0010	
DATA ADDRESS	ADDR	1111 01 1001	
SAVE STATE	SVST	1111 11 0001	
OVERLAY	OVLY	1111 11 0000	
PROFILE	PRFL	1111 10 1111	ENTRY NUMBER
PARITY ADDRESS	PADR	1111 11 0111	
EXECUTE	EXEC	1111 11 1110 010	
COMMUNICATE WITH GISMO	CWG	1111 11 1110 110	
ADD TIMER	ADDT	1111 11 1100 000	
SUBTRACT TIMER	SUBT	1111 11 1100 001	

APPENDIX C
SDL/UPL SYNTAX REFERENCE GUIDE

All of the railroad syntax diagrams previously used in this manual are also listed in this appendix. The SDL compiler railroad syntax diagrams are presented first, followed by the UPL compiler railroad syntax diagrams.

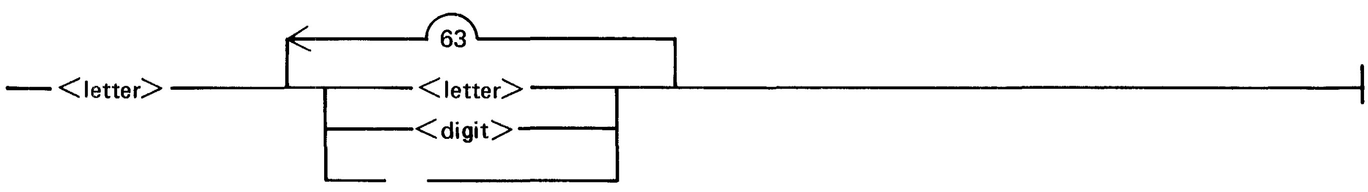
LISTING OF SDL RAILROAD SYNTAX DIAGRAMS

The railroad syntax diagrams valid for the SDL compiler are as follows:

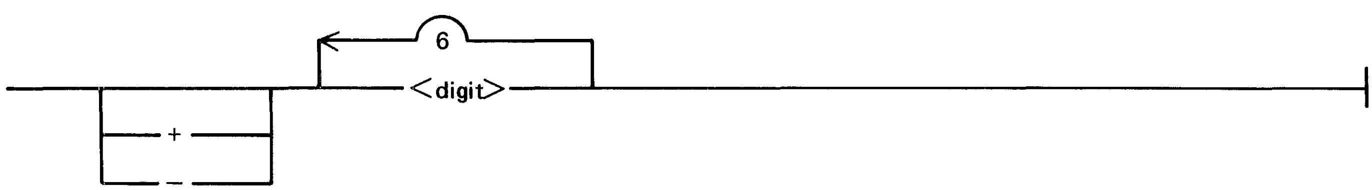
Fundamental Items

The following are the syntax diagrams for the fundamental items.

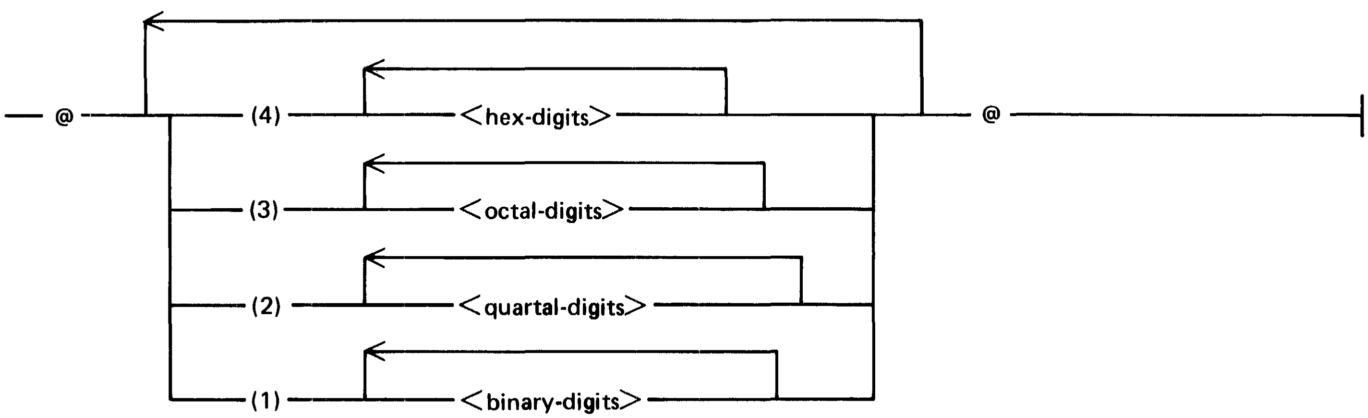
Identifiers



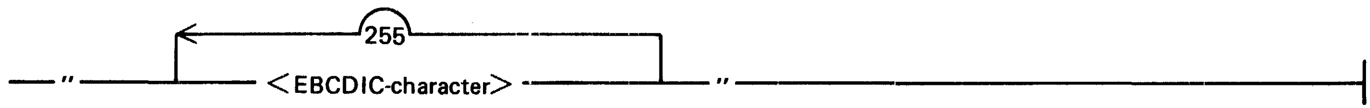
Numeric Literal



Bit-String Literal



Character-String Literal



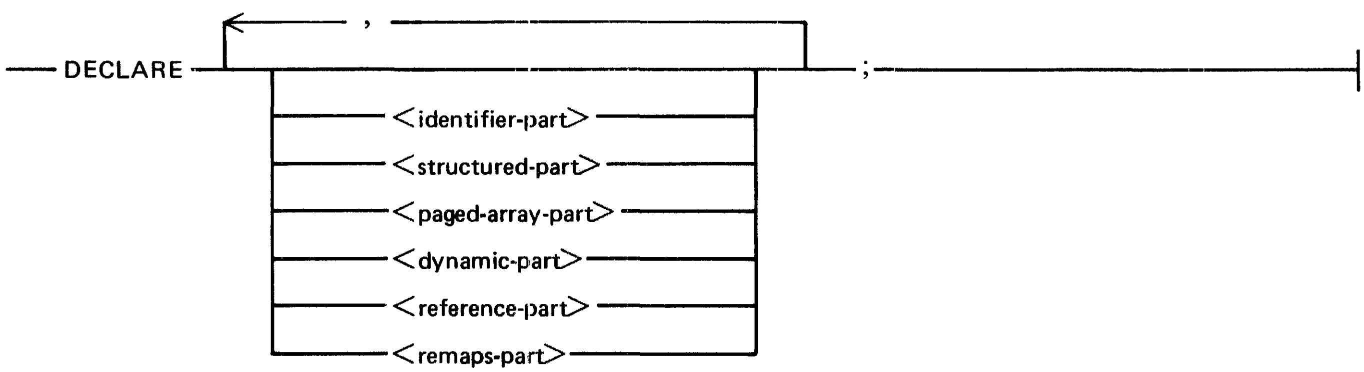
Enclosed Comment

End-of-Record Comment

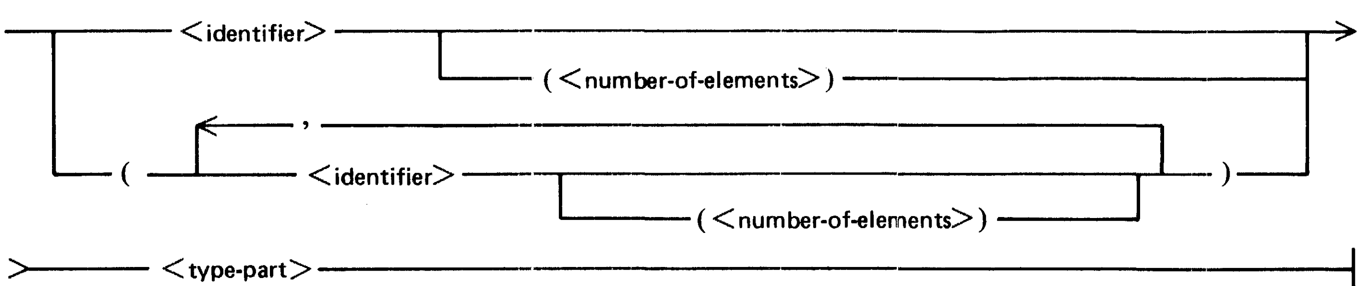
Declarations

The following are the syntax diagrams for the data, record, file, and switch-file declarations.

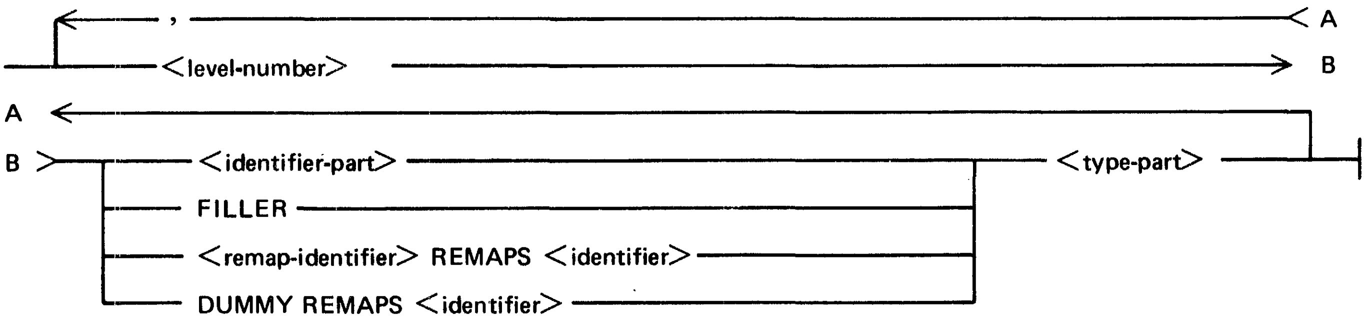
Data Declarations



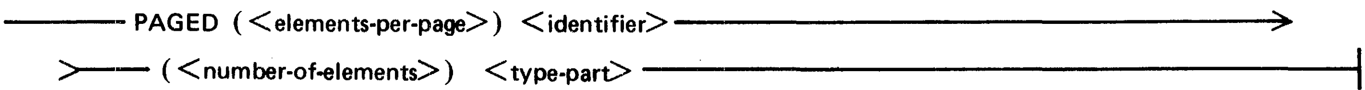
<identifier-part>



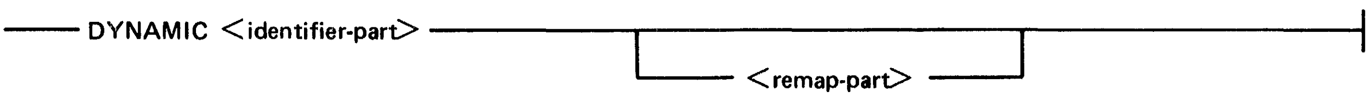
<structured-part>



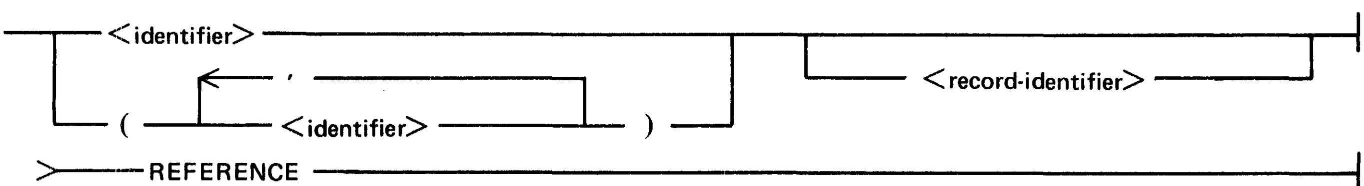
<paged-array-part>



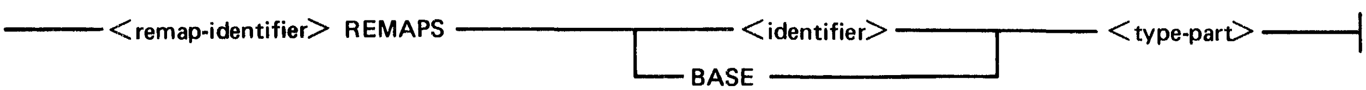
<dynamic-part>



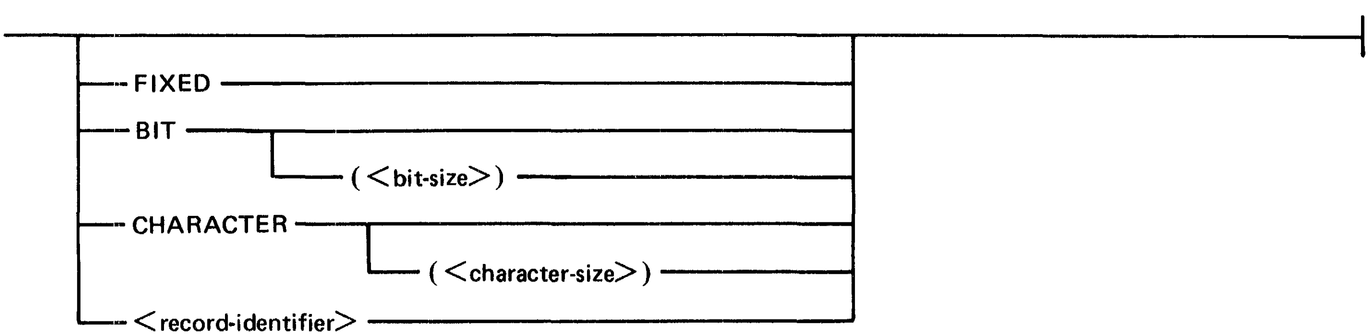
<reference-part>



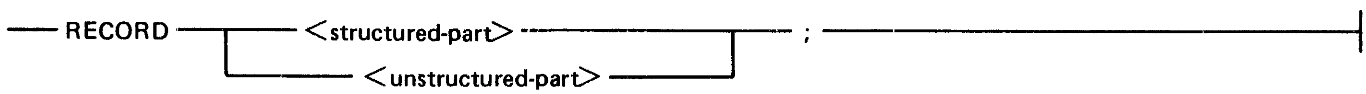
<remaps-part>



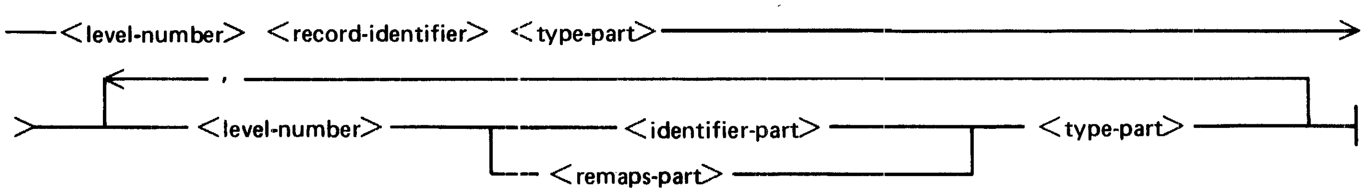
<type-part>



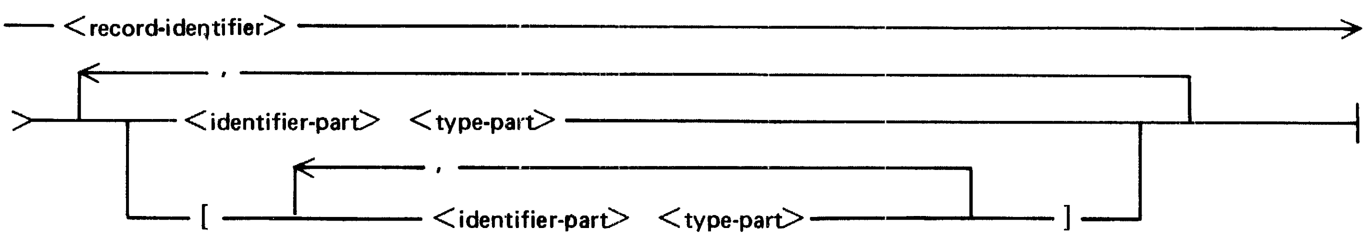
Record Declarations



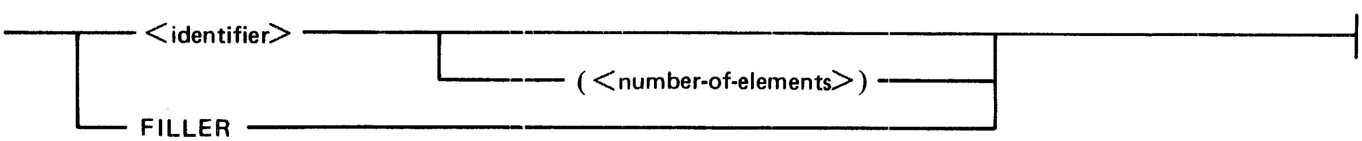
<structured-part>



<unstructured-part>

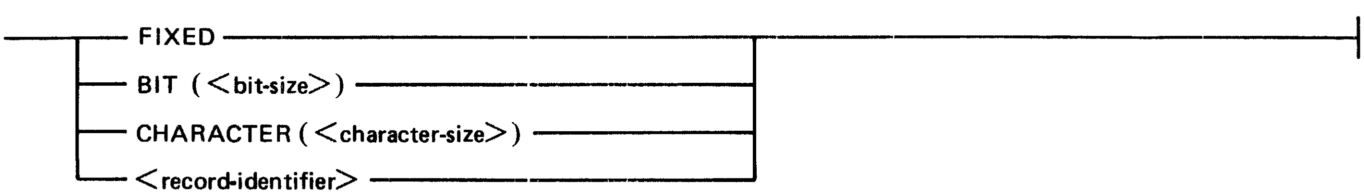


<identifier-part>

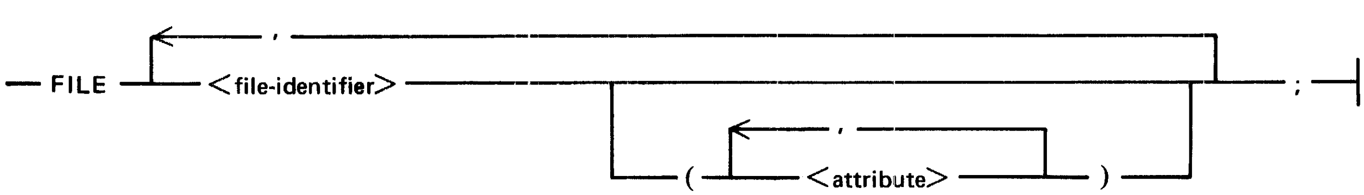


<remaps-part>

<type-part>



File Declarations

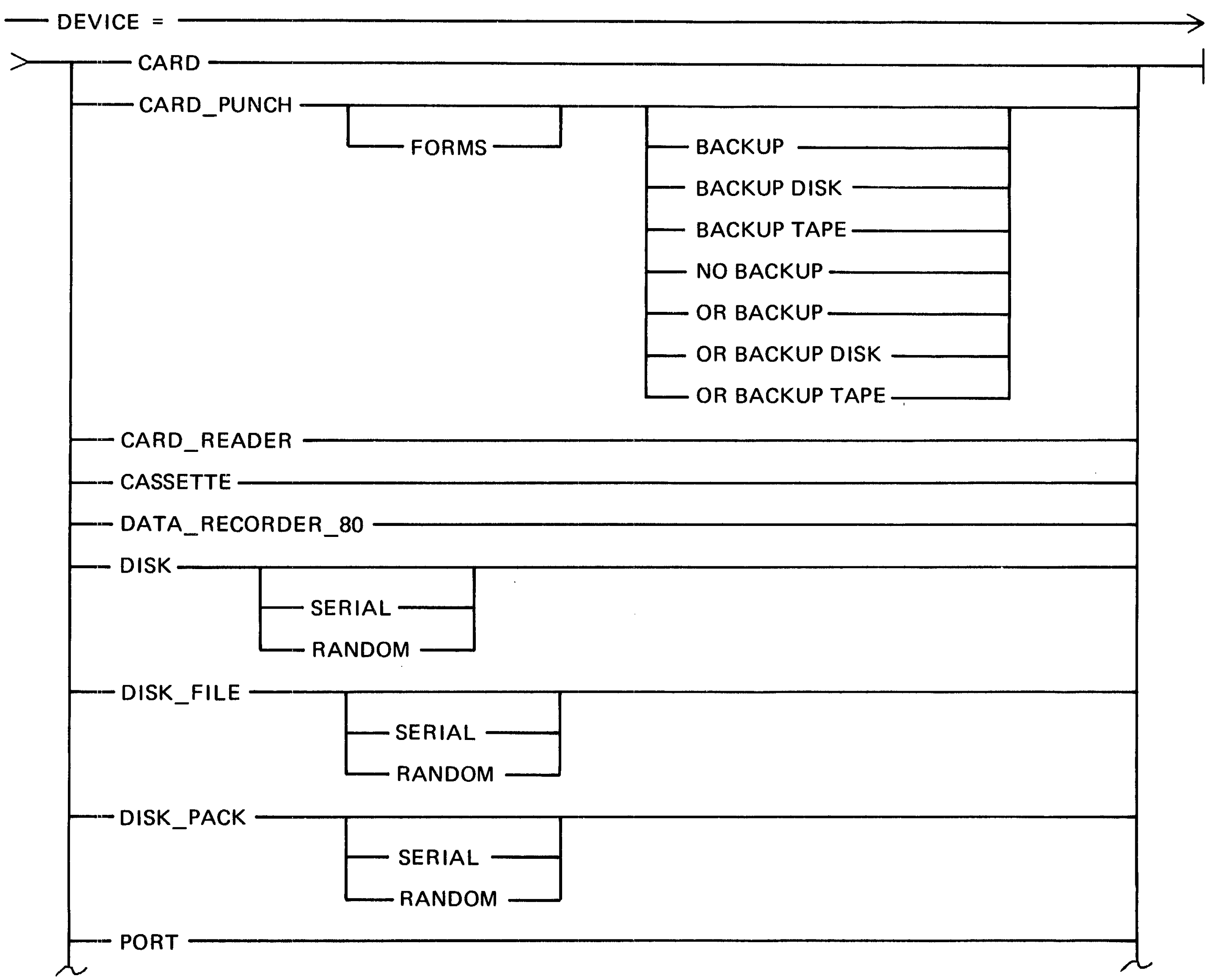


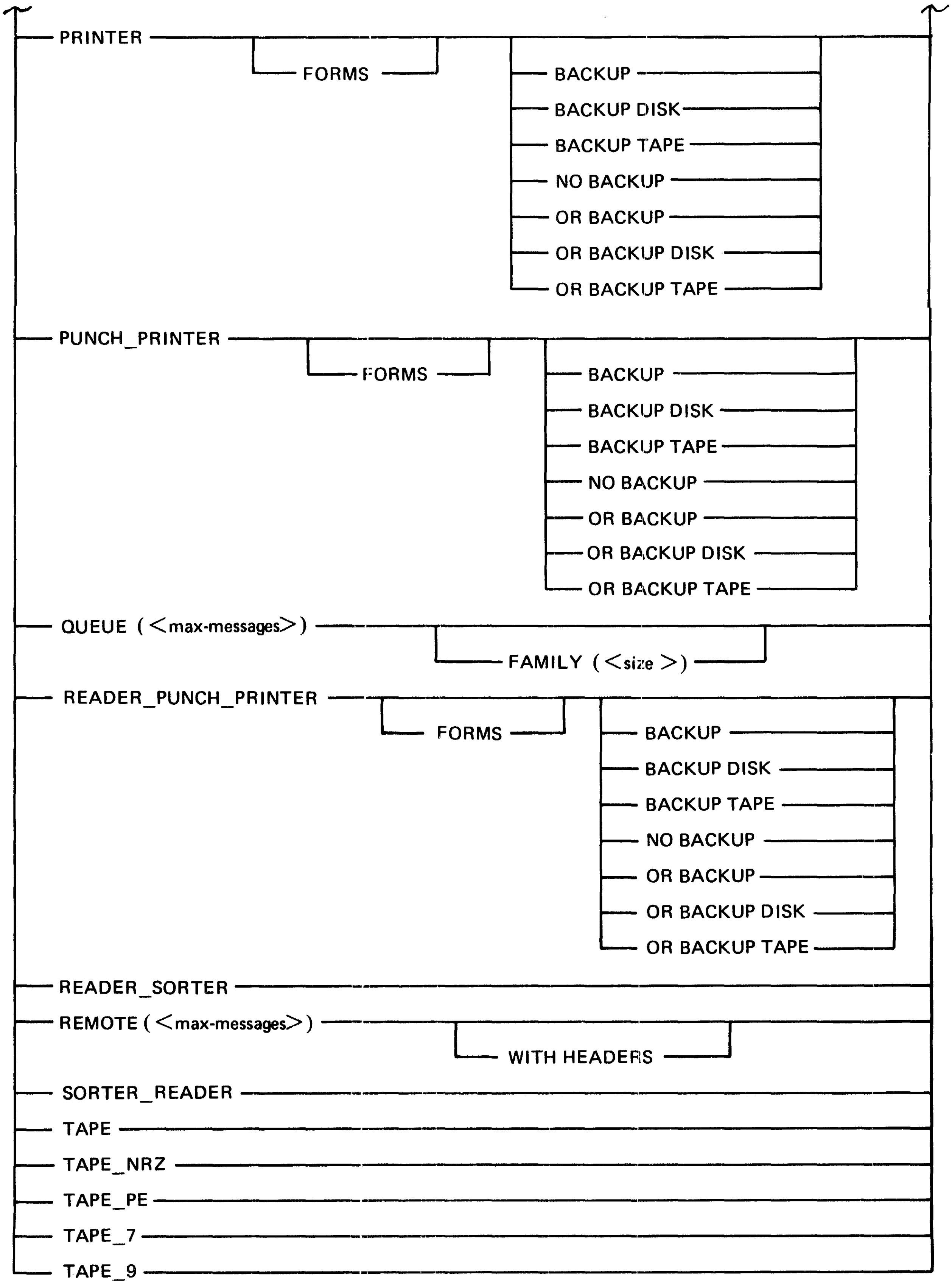
ALL_AREAS_AT_OPEN

AREAS

BUFFERS

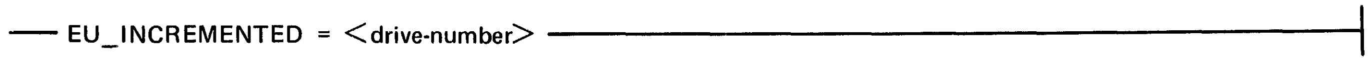
DEVICE




PRINTER
FORMS
BACKUP
BACKUP DISK
BACKUP TAPE
NO BACKUP
OR BACKUP
OR BACKUP DISK
OR BACKUP TAPE
PUNCH_PRINTER
FORMS
BACKUP
BACKUP DISK
BACKUP TAPE
NO BACKUP
OR BACKUP
OR BACKUP DISK
OR BACKUP TAPE
QUEUE (<max-messages>)
FAMILY (<size >)
READER_PUNCH_PRINTER
FORMS
BACKUP
BACKUP DISK
BACKUP TAPE
NO BACKUP
OR BACKUP
OR BACKUP DISK
OR BACKUP TAPE
READER_SORTER
REMOTE (<max-messages>)
WITH HEADERS
SORTER_READER
TAPE
TAPE_NRZ
TAPE_PE
TAPE_7
TAPE_9

END_OF_PAGE_ACTION

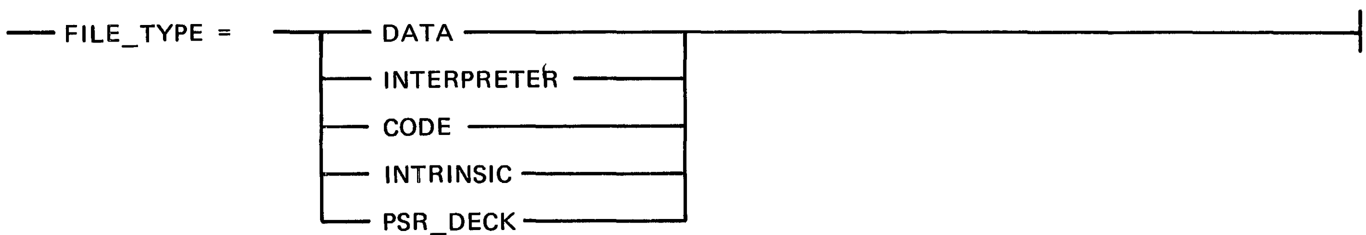
EU_INCREMENTED



EU_SPECIAL

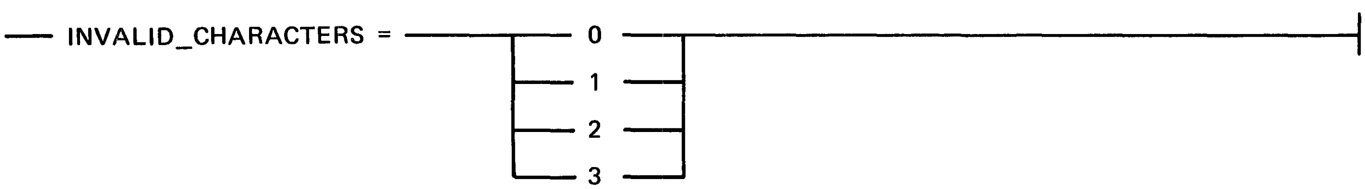
EXCEPTION_MASK

FILE_TYPE



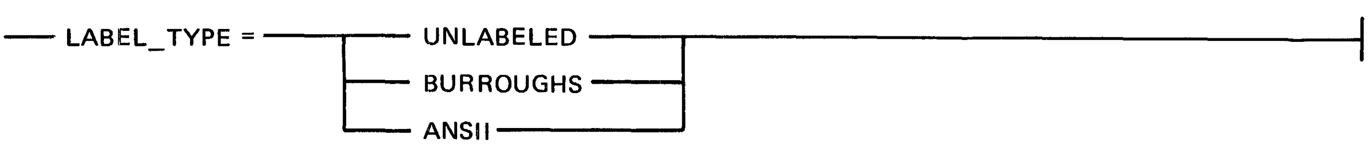
HOST_NAME

INVALID_CHARACTERS



LABEL

LABEL_TYPE



LOCK

```
── LOCK ──────────────────────────────────────────────────────┤
```

MODE

```
──MODE =──┬── ASCII ───┬──┬───────────┬──────────────────────┤
          ├── EBCDIC ──┤  ├── EVEN ───┤
          ├── BCL ─────┤  └── ODD ────┘
          └── BINARY ──┘
```

MULTI_PACK

```
── MULTI_PACK ────────────────────────────────────────────────┤
```

NUMBER_OF_STATIONS

```
── NUMBER_OF_STATIONS = <number> ─────────────────────────────┤
```

OPEN_OPTION

```
                     ┌←──── / ──────────┐
── OPEN_OPTION = ────┴──┬── INPUT ───┬──┴────────────────────┤
                        ├── OUTPUT ──┤
                        ├── NEW ─────┤
                        └── DEFAULT ─┘
```

OPTIONAL

```
── OPTIONAL ──────────────────────────────────────────────────┤
```

PACK_ID

```
── PACK_ID = "<pack-identifier>" ─────────────────────────────┤
```

PROTECTION

```
── PROTECTION = <number> ─────────────────────────────────────┤
```

PROTECTION_IO

```
── PROTECTION_IO = <number> ──────────────────────────────────┤
```

RECORDS

```
── RECORDS = ────┬── <physical-size> ──────────────────────┬──┤
                 └── <logical-size>/<records-per-block> ───┘
```

REEL

```
── REEL = <reel-number> ──────────────────────────────────────┤
```

REMOTE_KEY

```
── REMOTE_KEY ────────────────────────────────────────────────┤
```

SAVE

```
── SAVE = <number-of-days> ───────────────────────────────────┤
```

SECURITY

```
── SECURITYTYPE = <number> ───────────────────────────────────┤
```

SECURITYUSE

```
── SECURITYUSE = <number> ────────────────────────────────────┤
```

SERIAL

```
── SERIAL = ──────┬──── <number> ──────────────┬──────────────┤
                  └──── "<character-string>" ──┘
```

TRANSLATE

```
── TRANSLATE = "<file-identifier>" ───────────────────────────┤
```

USE_INPUT_BLOCKING

```
── USE_INPUT_BLOCKING ────────────────────────────────────────┤
```

USER_NAMED_BACKUP

```
── USER_NAMED_BACKUP ─────────────────────────────────────────┤
```

VARIABLE

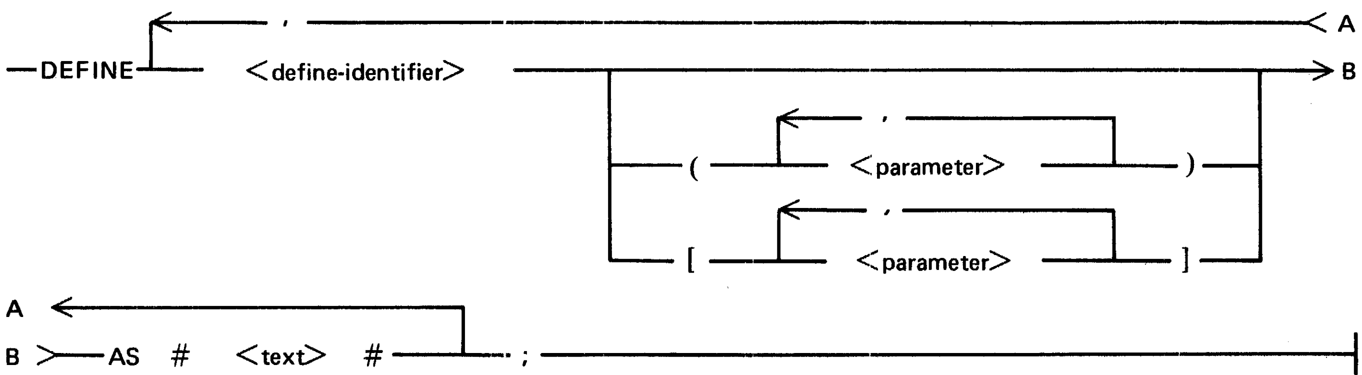
```
── VARIABLE ──────────────────────────────────────────────────┤
```

WORK_FILE

```
── WORK_FILE ─────────────────────────────────────────────────┤
```

Switch File Declarations

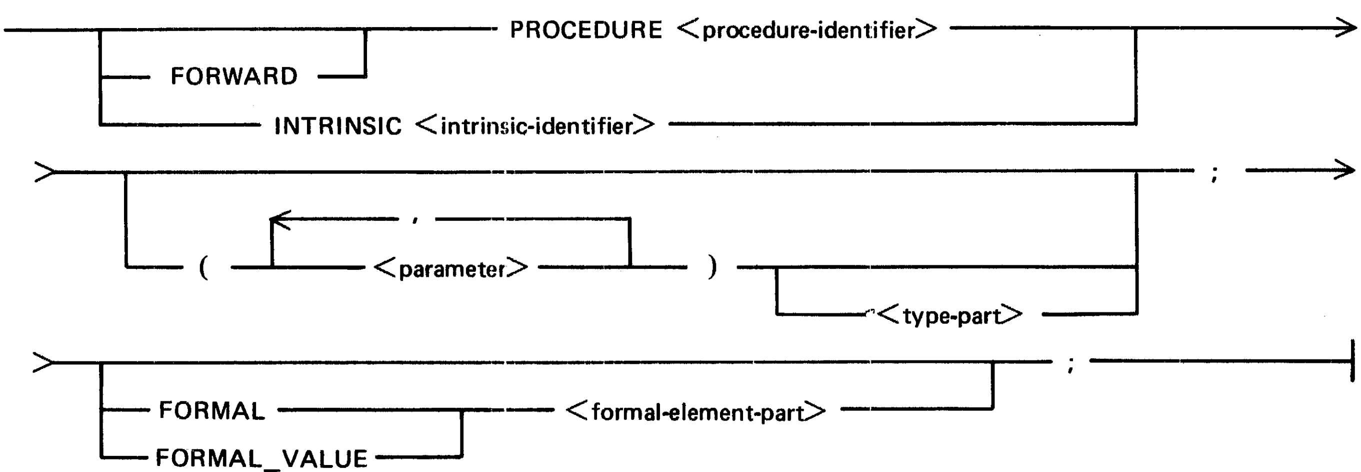
Define Statement



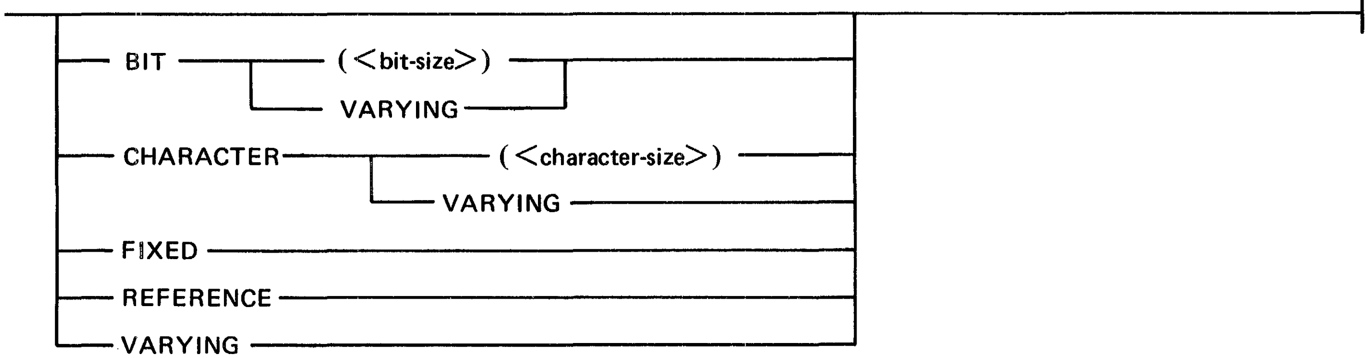
Procedure Statement

The following are the syntax diagrams for the procedure declaration, procedure body, procedure end, and procedure invocation statements.

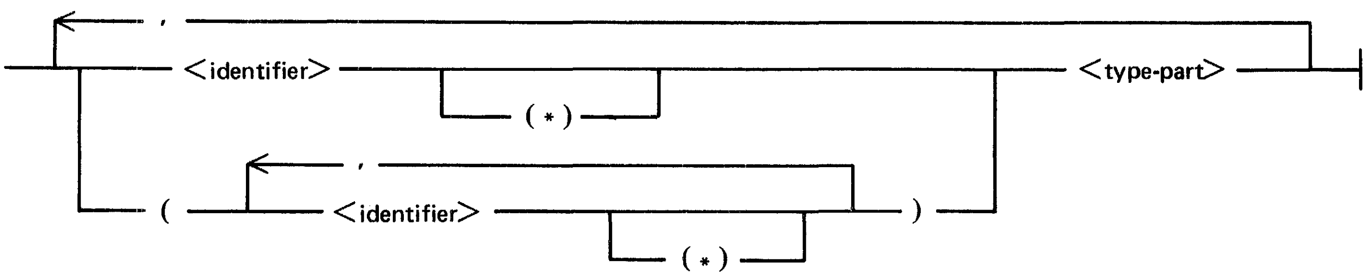
Procedure Declaration



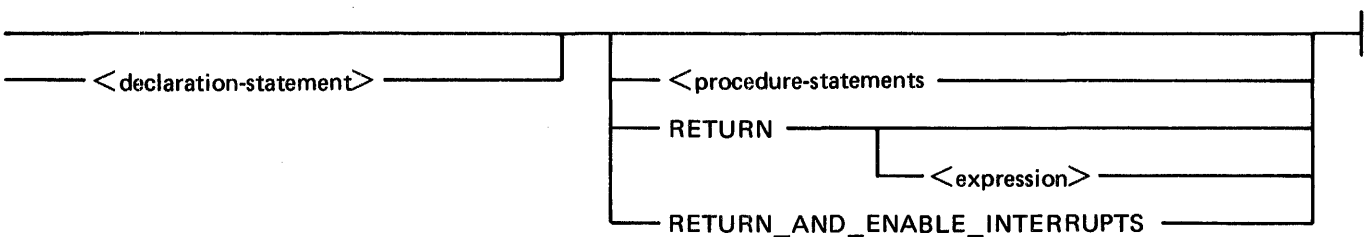
<type-part>



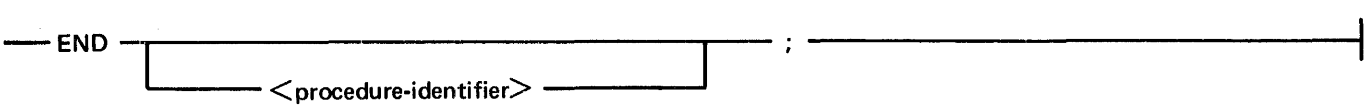
<formal-element-part>



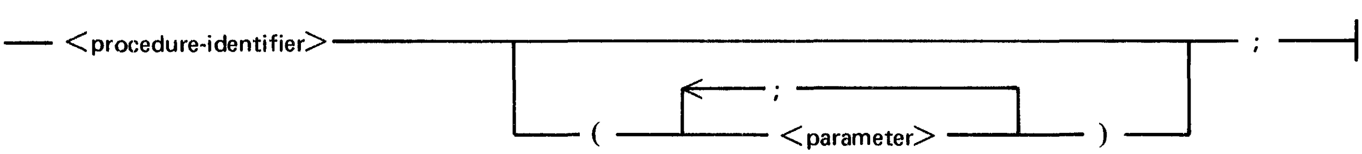
Procedure Body



Procedure End

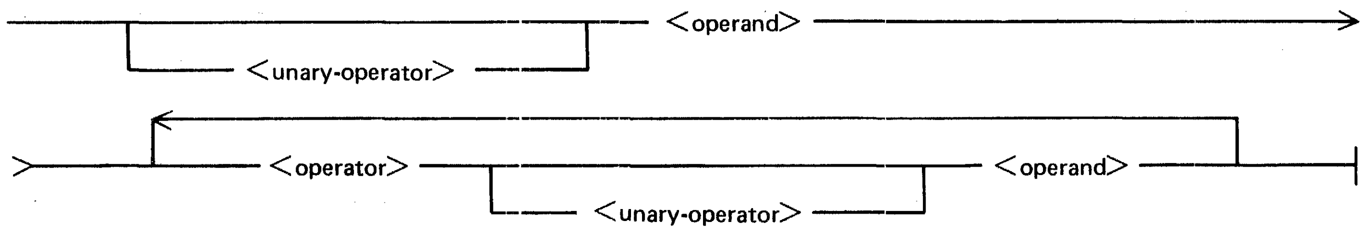


Procedure Invocations



Expressions

The following are the syntax diagrams for the expressions.

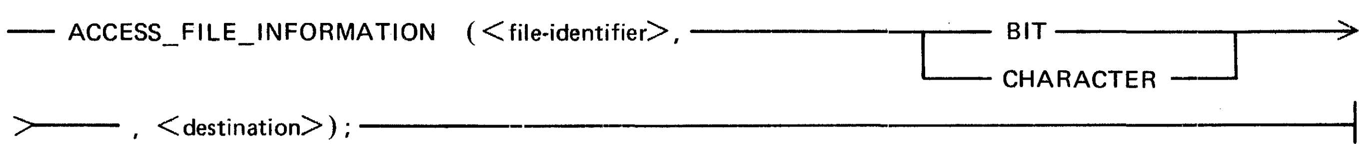


Verbs

The following are the syntax diagrams for the verbs.

ACCEPT

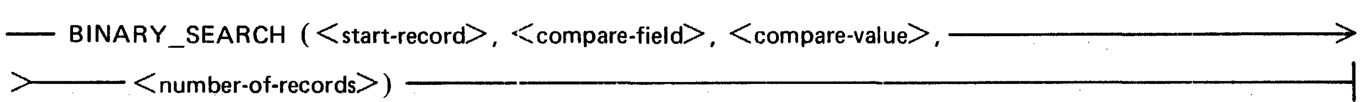
ACCESS_FILE_INFORMATION



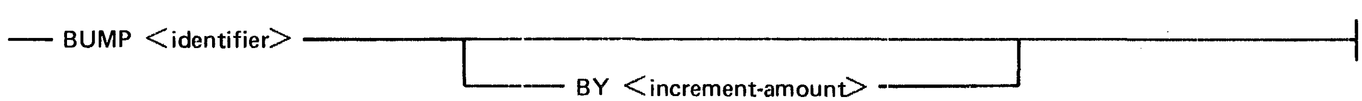
BASE_REGISTER

BINARY

BINARY_SEARCH



BUMP



CASE (format-1)

```
── CASE <index>; ──────────────────────────────────────────>
     >──── <statement-0>; ───────────────────────────────>
     >──── <statement-1>; ───────────────────────────────>
                 .
                 .
                 .
     >──── <statement-n>; ───────────────────────────────>
     >──── END CASE; ────────────────────────────────────┤
```

CASE (format-2

```
                              ┌<──── , ─────┐
── CASE <index> OF ( ─────────┴─ <expression> ─┴── ) ──────────┤
```

CHANGE

```
                                          ┌<──── , ───────────────┐
── CHANGE <file-identifier> TO ( ─────────┴─ <attribute> : = <value> ─┴── ); ──┤
```

CHAR_TABLE

```
                          ┌<──────── CAT ─────────────────┐
── CHAR_TABLE ( ──────────┴──┬─ "<EBCDIC-characters>" ──┬──┴── ) ──┤
                             └─ @<2-hexadecimal-numbers>@ ┘
```

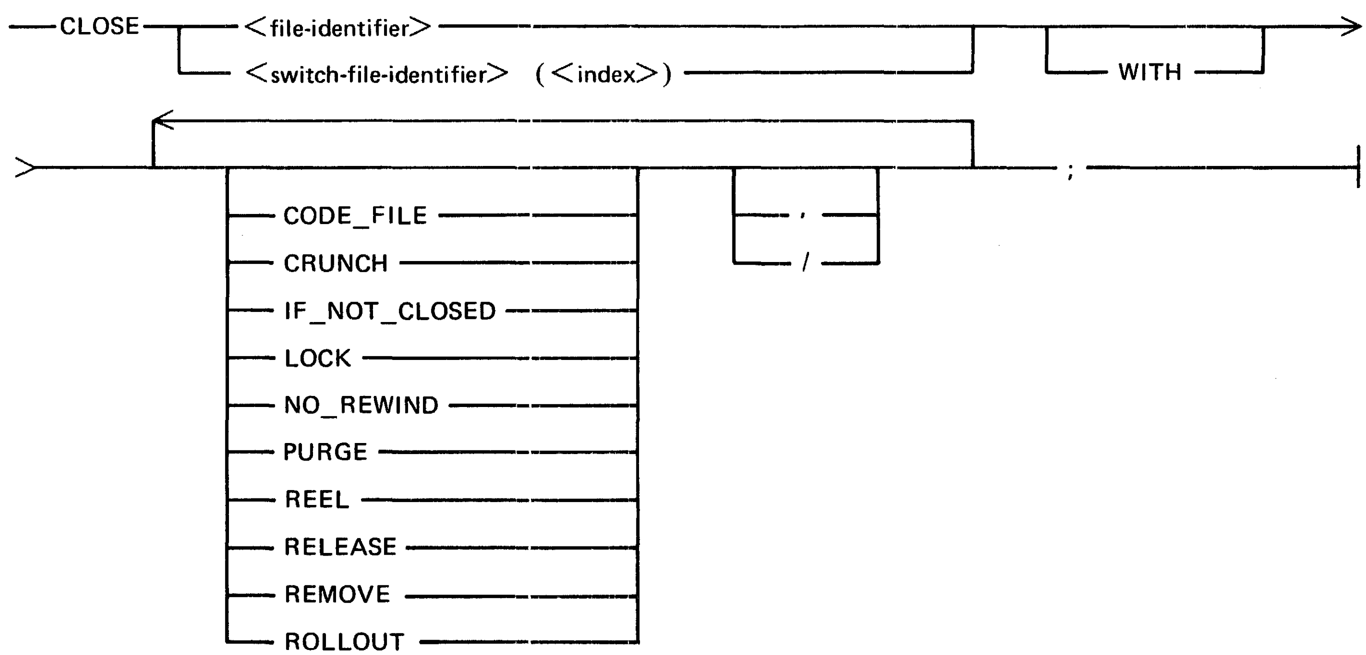
CHARACTER_FILL

```
── CHARACTER_FILL (<destination>,<source>); ─────────────────┤
```

CLEAR

```
                   ┌<──── , ──────────┐
── CLEAR ──────────┴─ <array-identifier> ─┴── ; ──────────────┤
```

CLOSE



COMMUNICATE_WITH_GISMO

— COMMUNICATE_WITH_GISMO (<communicate>) ; —|

COMMUNICATE

— COMMUNICATE (<MCP-communicate>) ; —|

COMPILE_CARD_INFO

— COMPILE_CARD_INFO (<destination>) ; —|

CONSOLE_SWITCHES

— CONSOLE_SWITCHES —|

CONTROL_STACK_BITS

— CONTROL_STACK_BITS —|

CONTROL_STACK_TOP

— CONTROL_STACK_TOP —|

CONVERT

```
── CONVERT (<convert-value>, ──┬─ FIXED ──────────────────────────────┬─ ) ──┤
                               ├─ BIT ─────────┬──────────────────────┤
                               └─ CHARACTER ───┘  └─ , ──┬─ 4 ─────────┤
                                                         ├─ 3 ─────────┤
                                                         ├─ 2 ─────────┤
                                                         └─ 1 ─────────┘
```

DATA_ADDRESS

```
── DATA_ADDRESS (<identifier>) ──────────────────────────────────────────┤
```

DATA_LENGTH

```
── DATA_LENGTH (<data-item>) ────────────────────────────────────────────┤
```

DATA_TYPE

```
── DATA_TYPE (<data-item>) ──────────────────────────────────────────────┤
```

DATE

```
── DATE ──┬──────────────────────────────────────────────────────┬───────┤
          └─ ( ──┬─ DAY ─────┬─ , ──┬─ BIT ─────────┬─ ) ─────────┘
                 ├─ JULIAN ──┤      ├─ CHARACTER ───┤
                 ├─ MONTH ───┤      └─ DIGIT ───────┘
                 └─ YEAR ────┘
```

DC_INITIATE_IO

```
── DC_INITIATE_IO (<port>, <channel>, <I/O-descriptor-address>); ────────┤
```

DEBLANK

```
── DEBLANK (<first-character>); ─────────────────────────────────────────┤
```

DECIMAL

```
── DECIMAL (<string>, <string-size>) ────────────────────────────────────┤
```

DECREMENT

```
── DECREMENT < identifier> ──────────┬──────────────────────────────┬──────┤
                                     └── BY < decrement-amount> ────┘
```

DELIMITED_TOKEN

```
── DELIMITED_TOKEN (<first-character-address> , <delimiters>, ──────────>
>──── <result-reference-identifier>) ──────────────────────────────────────┤
```

DESCRIPTOR

```
── DESCRIPTOR ( ──┬── <simple-identifier> ──┬── ) ─────────────────┤
                  └── <array-identifier> ───┘
```

DISABLE_INTERRUPTS

```
── DISABLE_INTERRUPTS ; ───────────────────────────────────────────┤
```

DISPATCH

```
── DISPATCH (<port-and-channel>, <I/O-descriptor-address>) ────────┤
```

DISPLAY

```
── DISPLAY (<display-identifier>) ──────┬────────────────┬── ; ────┤
                                        └── , CRUNCHED ──┘
```

DISPLAY_BASE

```
── DISPLAY_BASE ───────────────────────────────────────────────────┤
```

DO

```
                                                  ┌<──────────────────┐
──DO ─┬──────────────────┬┬───────────┬── ; ──────┴┬─────────────────┬┴──>
      └── <group-name> ──┘└─ FOREVER ─┘            └─ <statement> ; ─┘
>──── END ──┬──────────────────┬── ; ──────────────────────────────────┤
            └── <group-name> ──┘
```

DUMP_FOR_ANALYSIS

```
── DUMP_FOR_ANALYSIS; ─────────────────────────────────────────────┤
```

DYNAMIC_MEMORY_BASE

```
── DYNAMIC_MEMORY_BASE ──────────────────────────────────┤
```

ENABLE_INTERRUPTS

```
── ENABLE_INTERRUPTS; ───────────────────────────────────┤
```

ENTER_COROUTINE

```
── ENTER_COROUTINE (<coroutine-table>); ─────────────────┤
```

ERROR_COMMUNICATE

```
── ERROR_COMMUNICATE (<error-message>); ─────────────────┤
```

EVALUATION_STACK_TOP

```
── EVALUATION_STACK_TOP ─────────────────────────────────┤
```

EXECUTE

```
                ┌←──────── , ─────────┐
──EXECUTE ( ────┴─── <operation-list> ─┴── ) ───────────┤
```

EXIT_COROUTINE

```
── EXIT_COROUTINE (<coroutine-table>); ──────────────────┤
```

FETCH

```
── FETCH (<I/O-reference-address>, <port-and-channel-address>, ───→
>──── <result-descriptor-address>); ─────────────────────┤
```

FETCH_COMMUNICATE_MSG_PTR

```
── FETCH_COMMUNICATE_MSG_PTR ────────────────────────────┤
```

FIND_DUPLICATE_CHARACTERS

```
── FIND_DUPLICATE_CHARACTERS (<reference-identifier-1>, ───→
>──── <count-identifier>, <character-identifier>, ─────────→
>──── <reference-identifier-2>); ────────────────────────┤
```

FINI

```
── FINI ─────────────────────────────────────────────────┤
```

FREEZE_PROGRAM

```
── FREEZE_PROGRAM; ──────────────────────────────────────┤
```

GROW

```
── GROW (<paged-array-identifier>, <increase-amount>) ; ──┤
```

HALT

```
── HALT (<halt-value>) ; ─────────────────────────────────┤
```

HASH_CODE

```
── HASH_CODE (<hash-code-value>) ─────────────────────────┤
```

IF, THEN, and ELSE (Conditionals)

```
── IF <condition> THEN ──┬─────────────────┬─ ; ─────────>
                         └─ <statement-1> ─┘

>───┬──────────────────────────────────────┬──────────────┤
    └─ ELSE ──┬─────────────────┬─ ; ──────┘
              └─ <statement-2> ─┘
```

INITIALIZE_VECTOR

```
── INITIALIZE_VECTOR (<table-address>) ; ─────────────────┤
```

LAST_LIO_STATUS

```
── LAST_LIO_STATUS ───────────────────────────────────────┤
```

LENGTH

```
── LENGTH (<identifier>) ─────────────────────────────────┤
```

LIMIT_REGISTER

```
── LIMIT_REGISTER ────────────────────────────────────────┤
```

LOCATION

```
── LOCATION ( ──┬─ <identifier> ────────────┬─ ) ─────────┤
                ├─ <array-identifier> ──────┤
                └─ <procedure-identifier> ──┘
```

MAKE_DESCRIPTOR

— MAKE_DESCRIPTOR (<descriptor>) —

MAKE_READ_ONLY

— MAKE_READ_ONLY (<paged-array-identifier>), <page-number>) ; —

MAKE_READ_WRITE

— MAKE_READ_WRITE (<paged-array-identifier>, <page-number>), —

MESSAGE_COUNT

— MESSAGE_COUNT (<queue-file-id>, <identifier>) ; —

MONITOR

— MONITOR — [AND NOT | OR | + | – | ,] — [$ALL | $NONE | <sequence-range> | <procedure-name>] — ;

M_MEM_SIZE

— M_MEM_SIZE —

NAME_OF_DAY

— NAME_OF_DAY —

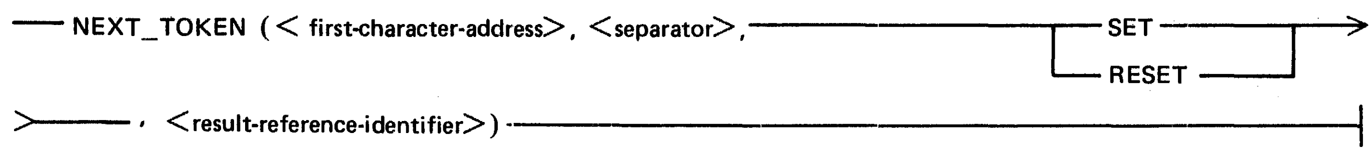
NAME_STACK_TOP

— NAME_STACK_TOP —

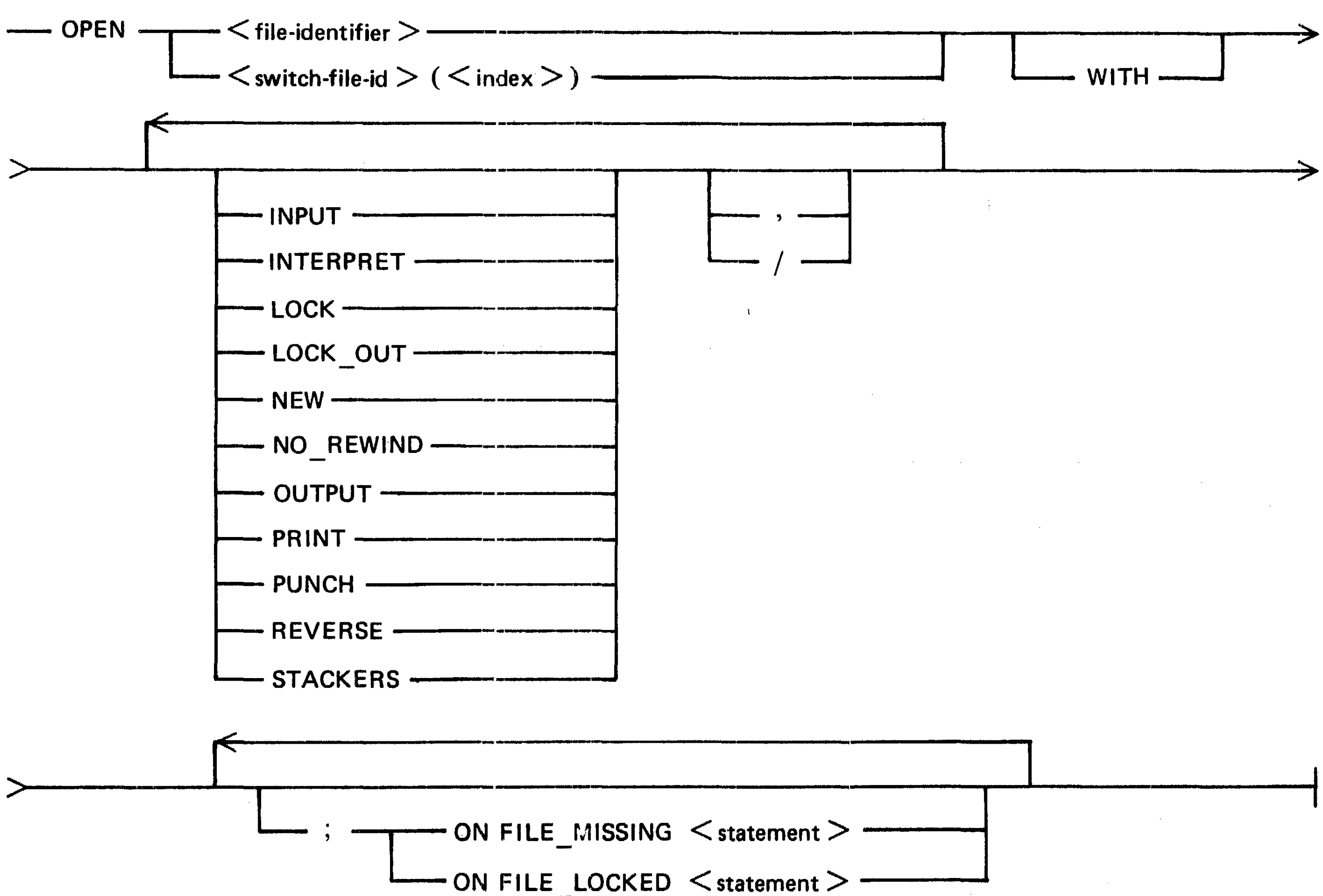
NEXT_ITEM

— NEXT_ITEM (<identifier>) —

NEXT_TOKEN



OPEN



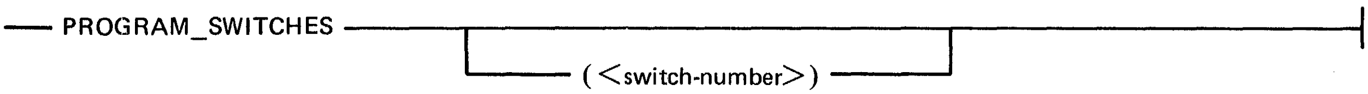
OVERLAY

PARITY_ADDRESS

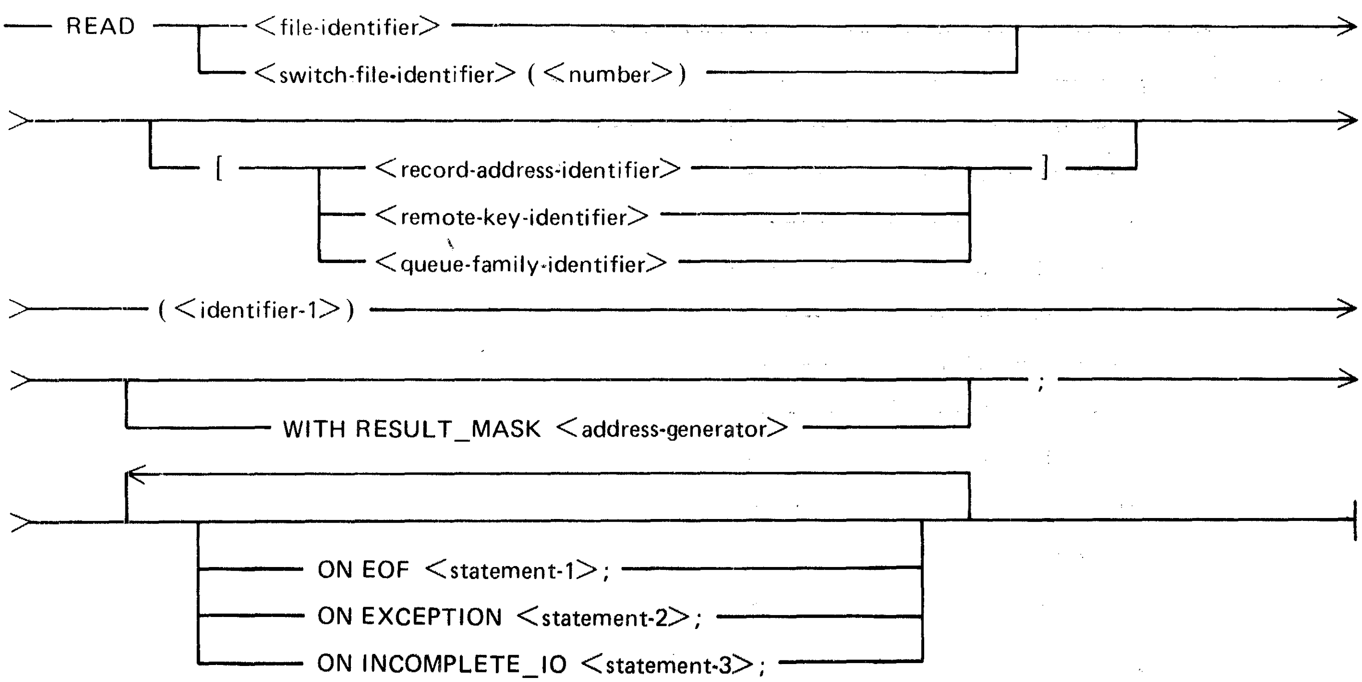
PREVIOUS_ITEM

PROCESSOR_TIME

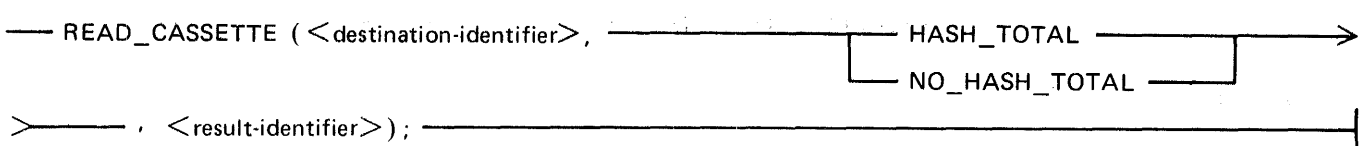
PROGRAM_SWITCHES



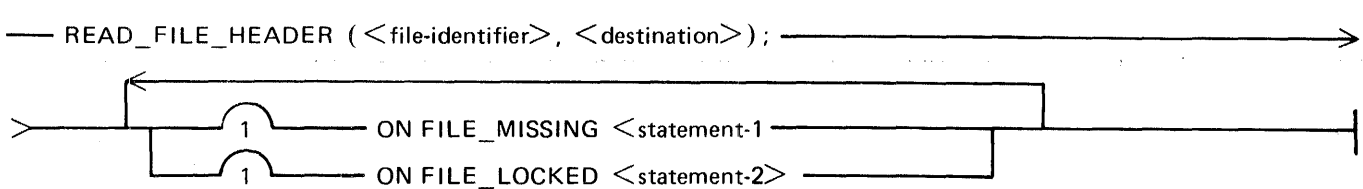
READ



READ_CASSETTE



READ_FILE_HEADER



READ_FPB

```
── READ_FPB ( ──┬── <file-identifier> ──┬── , <destination> ) ; ──────┤
                └── <file-number> ──────┘
```

READ_OVERLAY

```
── READ_OVERLAY ( <overlay-information> ) ; ──────────────────────────┤
```

REDUCE

```
── REDUCE <reference-identifier-1> ────────────────────────────────────>

>──┬────────────────────────────────────────┬── UNTIL ─────────────────>
   └── SETTING <reference-identifier-2> ────┘

>──┬── FIRST ──┬──┬── EQL ──┬──┬── <literal> ─────┬── ; ──────────────>
   └── LAST ───┘  ├── NEQ ──┤  └── <identifier> ──┤
                  ├── = ────┤                     │
                  ├── /= ───┘                     │
                  └── IN <character-table-identifier> ──┘

>──┬──────────────────────────────────┬───────────────────────────────┤
   ├── ON EOS_CYCLE <statement-1> ; ──┤
   └── ON EOS <statement-2> ; ────────┘
```

REFER

```
── REFER <reference-identifier> TO <identifier> ; ────────────────────┤
```

REFER_ADDRESS

```
── REFER_ADDRESS ( <reference-identifier>, <address> ) ; ─────────────┤
```

REFER_LENGTH

```
── REFER_LENGTH ( <reference-identifier>, <length> ) ; ───────────────┤
```

REFER_TYPE

```
── REFER_TYPE ( <reference-identifier>, <type> ) ; ───────────────────┤
```

RESTORE

```
                ┌←──── , ─────┐
── RESTORE ( ───┴── <value> ──┴── ) ; ────────────────────────────────┤
```

RETURN

```
── RETURN ─┬──────────────────┬─ ; ──────────────────────────┤
           └── <expression> ──┘
```

RETURN_AND_ENABLE_INTERRUPTS

```
── RETURN_AND_ENABLE_INTERRUPTS: ─────────────────────────────┤
```

REVERSE_STORE

```
── REVERSE_STORE (<address-generator-1>, <address-generator-2>, ──────>
>──── < - - - - > , <address-generator-n>, <expression>) ─────────────┤
```

SAVE

```
            ┌←──────── , ────────┐
── SAVE ( ──┴──── <value> ───────┴── ); ─────────────────────────┤
```

SAVE_STATE

```
── SAVE_STATE; ───────────────────────────────────────────────────┤
```

SEARCH_DIRECTORY

```
── SEARCH_DIRECTORY (<file-identifier, <identifier>, ─────────────────>
>─┬─ BIT ─────────┬─ ); ─┬────────────────────────────────────────┬──>
  └─ CHARACTER ───┘      └── ON FILE_MISSING <statement-1>; ──────┘
>──┬───────────────────────────────────────┬──────────────────────┤
   └── ON FILE_LOCKED <statement-2>; ──────┘
```

SEARCH_LINKED_LIST

```
── SEARCH_LINKED_LIST (<first-item>, <compare-field>, ────────────────>
>── <compare-value>, <relation>, <link-field>); ──────────────────────┤
```

SEARCH_SDL_STACKS

```
── SEARCH_SDL_STACKS (<stack-base>, <stack-top>, <compare-base>, ─────>
>──── <compare-top>) ─────────────────────────────────────────────────┤
```

SEARCH_SERIAL_LIST

```
— SEARCH_SERIAL_LIST (<compare-value>, <relation>, <compare-field> ——>
>—— , <first-item>, <table-length>, <result-identifier>); ——|
```

SEEK

```
— SEEK <file-identifier> [<record-address-identifier>] ; ——|
```

SEGMENT_PAGE

```
— SEGMENT_PAGE (<segment-identifier> ——>

        <———————(1)————————
>———┬——— , IMPORTANT ————————┬——— ); ——|
    └——— OF <page-identifier> ——┘
```

SKIP

```
— SKIP <file-identifier> TO <channel-number>; ——|
```

SORT

```
— SORT (<sort-information-table>, <key-table>, ——>
>—— <input-file-identifier>, <output-file-identifier> ——>
>———┬————————————————————————————————┬——— ); ——|
    └——— , <translate-file-identifier> ——┘
```

SORT_MERGE

```
— SORT_MERGE (<sort-information-table>, <key-table>, ——>
>—— <merge-input-table>, <output-file-identifier> ——>
>———┬————————————————————————————————┬——— ); ——|
    └——— , <translate-file-identifier> ——┘
```

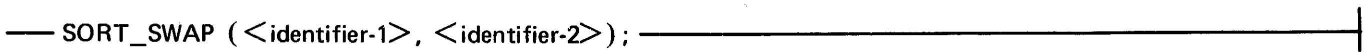
SORT_SEARCH

```
— SORT_SEARCH (<first-table-entry-address>, <limit>); ——|
```

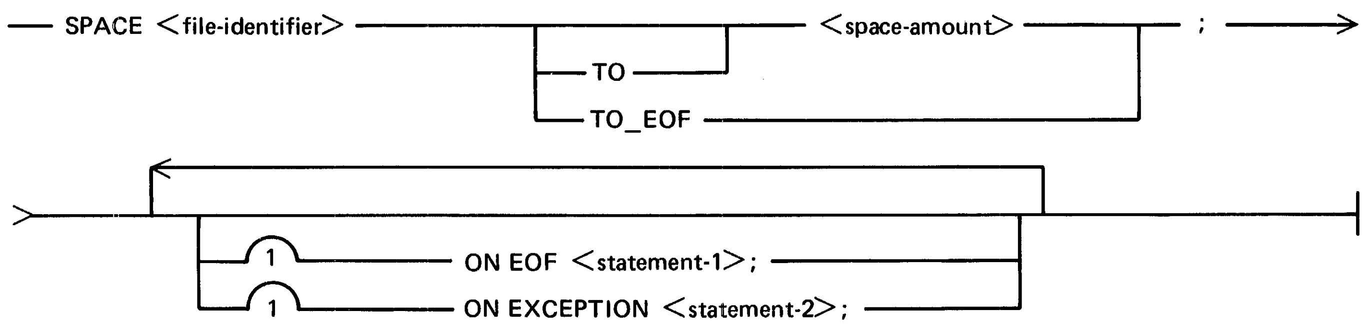
SORT_STEP_DOWN

```
— SORT_STEP_DOWN (<record-1>, <record-2>, <key-table-address>); ——|
```

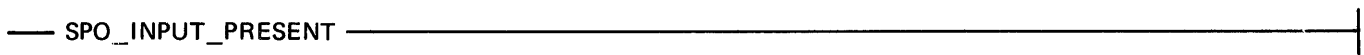
SORT__SWAP



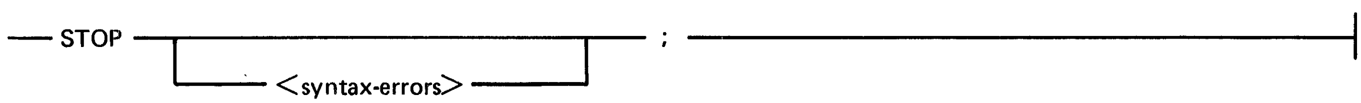
SPACE



SPO__INPUT__PRESENT



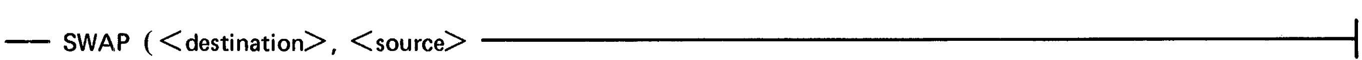
STOP



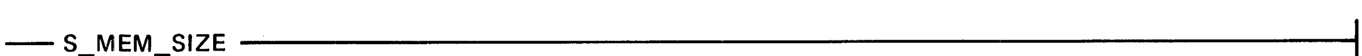
SUBBIT

SUBSTR

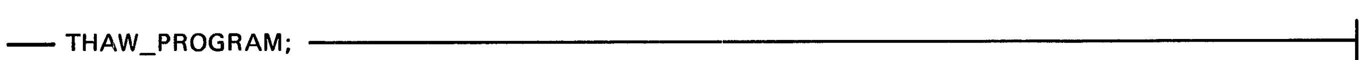
SWAP



S__MEM__SIZE



THAW__PROGRAM



THREAD_VECTOR

```
── THREAD_VECTOR (<table-address>, <index>); ──────────────┤
```

TIME

```
──TIME──┬─────────────────────────────────────────────────────────┬──┤
        └── ( ──┬── CIVILIAN ──┬── , ──┬── BIT ────────┬── ) ──┘
                ├── COUNTER ───┤       ├── CHARACTER ──┤
                └── MILITARY ──┘       └── DIGIT ──────┘
```

TIMER

```
── TIMER ──────────────────────────────────────────────────┤
```

TRACE

```
──────────┬── TRACE ──┬───────────────────────┬── ; ──────┤
── NO ────┘           └── (<trace-options>) ──┘
```

TRANSLATE

```
── TRANSLATE (<source-identifier>, <source-item-size>, ──────────>
>──── <translate-table>, <translate-item-size>, <result-identifier>) ──┤
```

UNDO

```
── UNDO ──┬──────────────────┬── ; ───────────────────────┤
          └── <identifier> ──┘
```

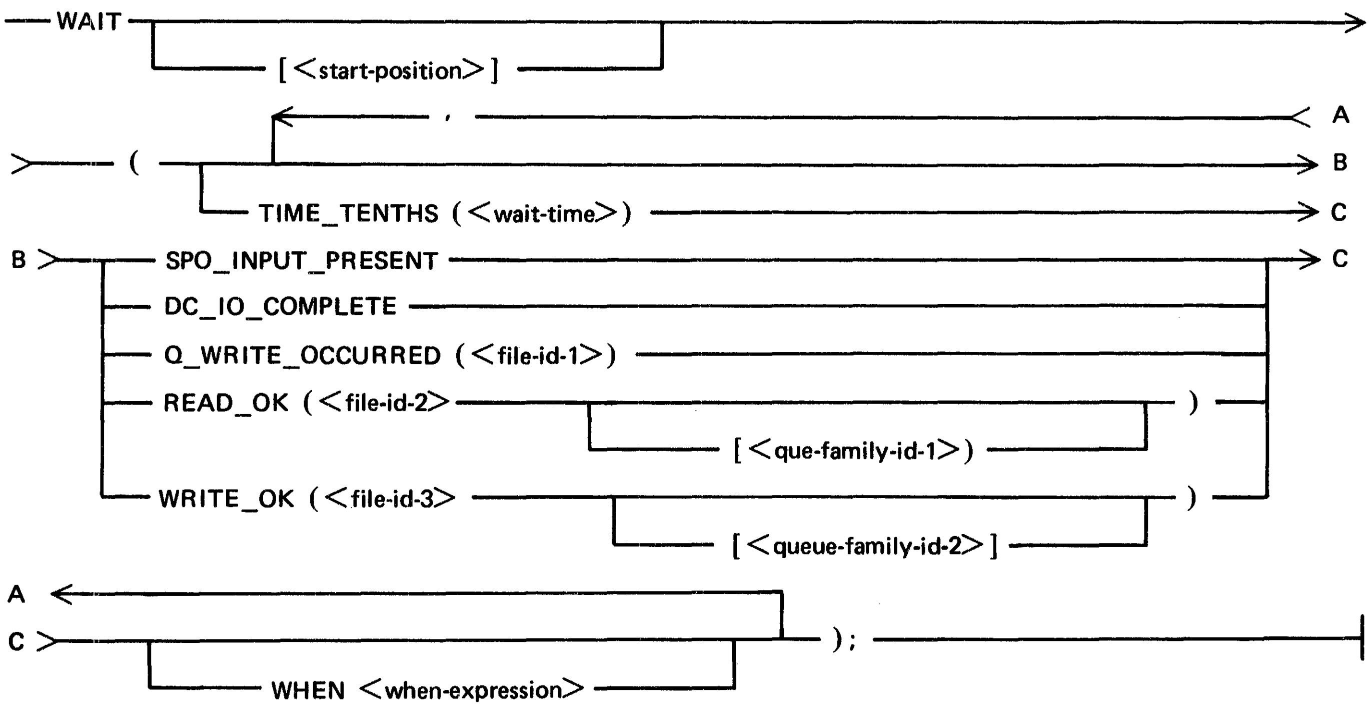
USE

```
             ┌←──── , ─────────────────┐
── USE ( ────┴── <declared-identifier> ─┴──── ) OF <defined-identifier>; ────┤
```

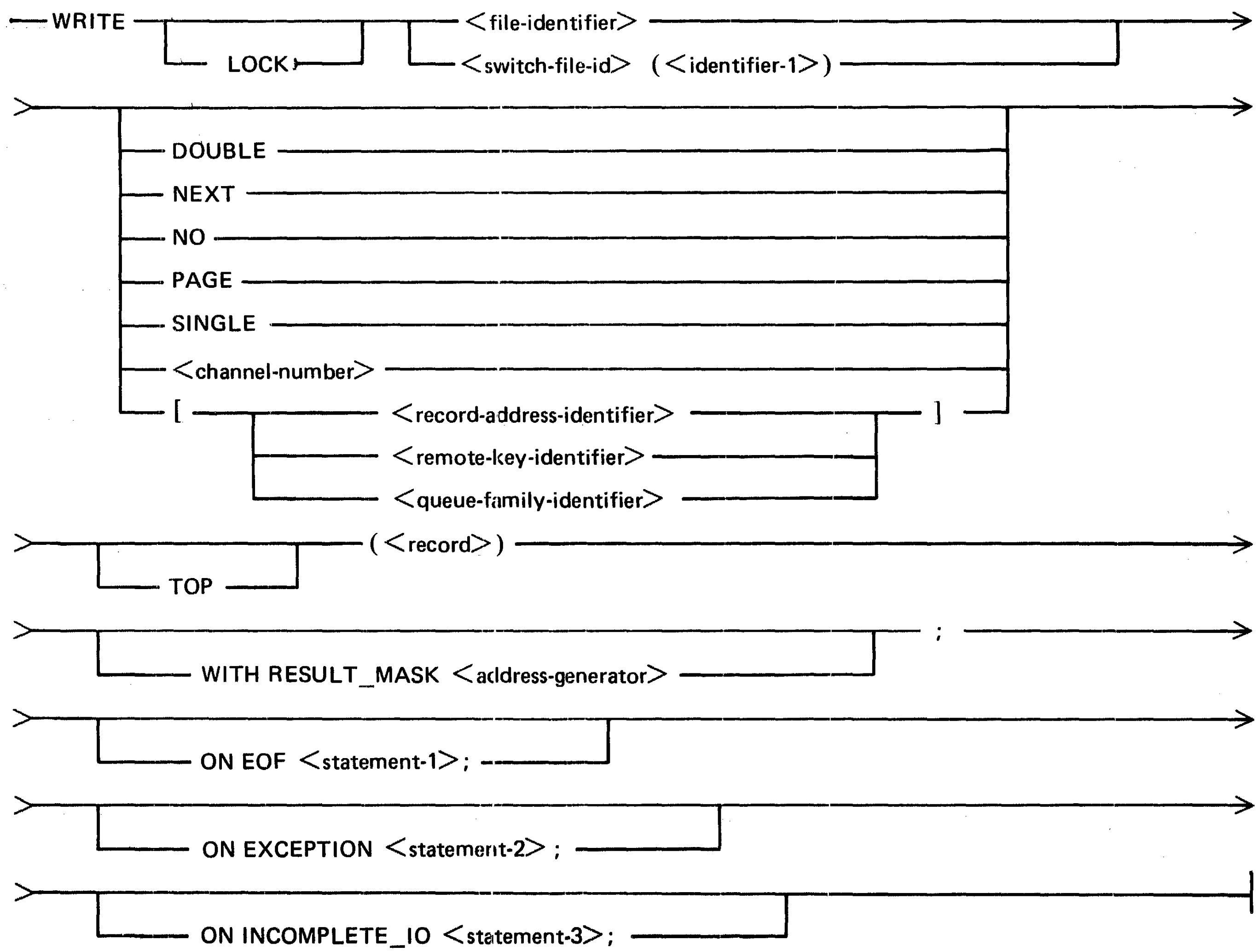
VALUE_DESCRIPTOR

```
── VALUE_DESCRIPTOR (<address-field>); ────────────────────┤
```

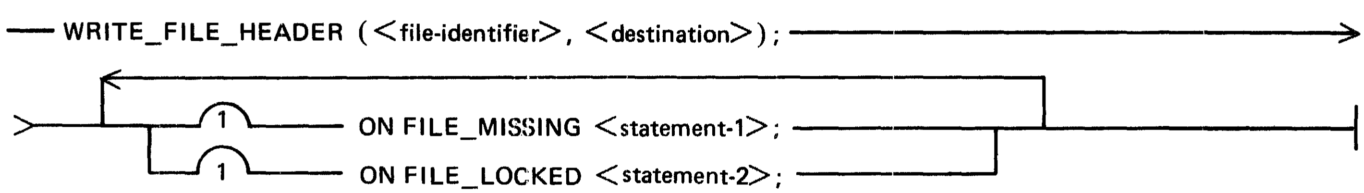
WAIT


WAIT
[<start-position>]
,
A
(
B
TIME_TENTHS (<wait-time>)
C
B
SPO_INPUT_PRESENT
C
DC_IO_COMPLETE
Q_WRITE_OCCURRED (<file-id-1>)
READ_OK (<file-id-2>
)
[<que-family-id-1>)
WRITE_OK (<file-id-3>
)
[<queue-family-id-2>]
A
C
) ;
WHEN <when-expression>

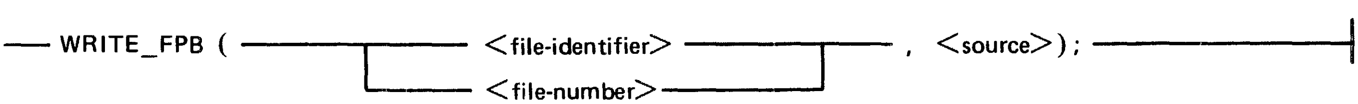
WRITE



WRITE_FILE_HEADER



WRITE_FPB



WRITE_OVERLAY

```
—— WRITE_OVERLAY (<overlay-information>); ——————————————————————————————————|
```

X_ADD

```
—— X_ADD (<expression-1>, <expression-2>) ——————————————————————————————————|
```

X_DIV

```
—— X_DIV (<expression-1>, <expression-2>) ——————————————————————————————————|
```

X_MOD

```
—— X_MOD (<expression-1>, <expression-2>) ——————————————————————————————————|
```

X_MUL

```
—— X_MUL (<expression-1>, <expression-2>) ——————————————————————————————————|
```

X_SUB

```
—— X_SUB (<expression-1>, <expression-2>) ——————————————————————————————————|
```

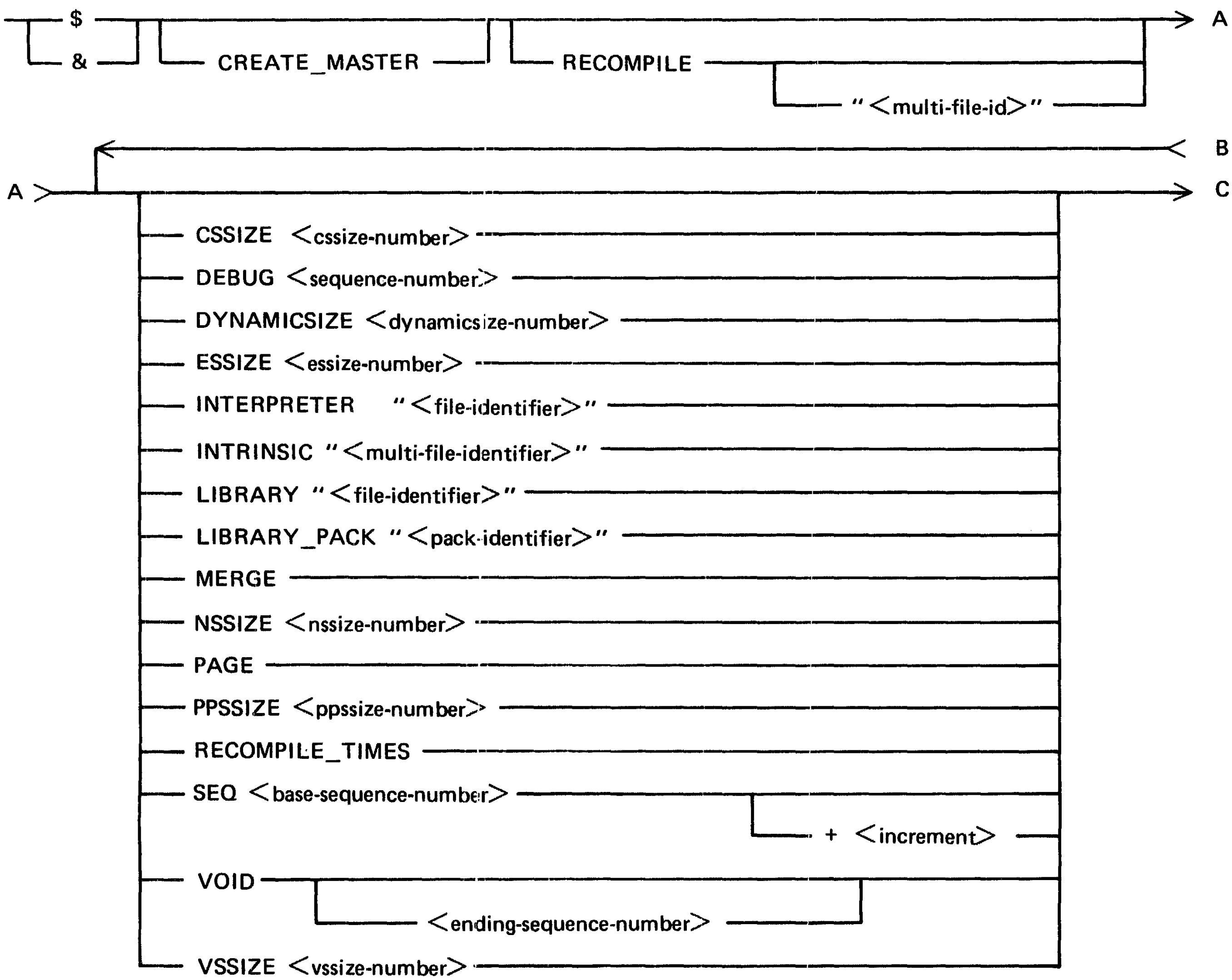
ZIP

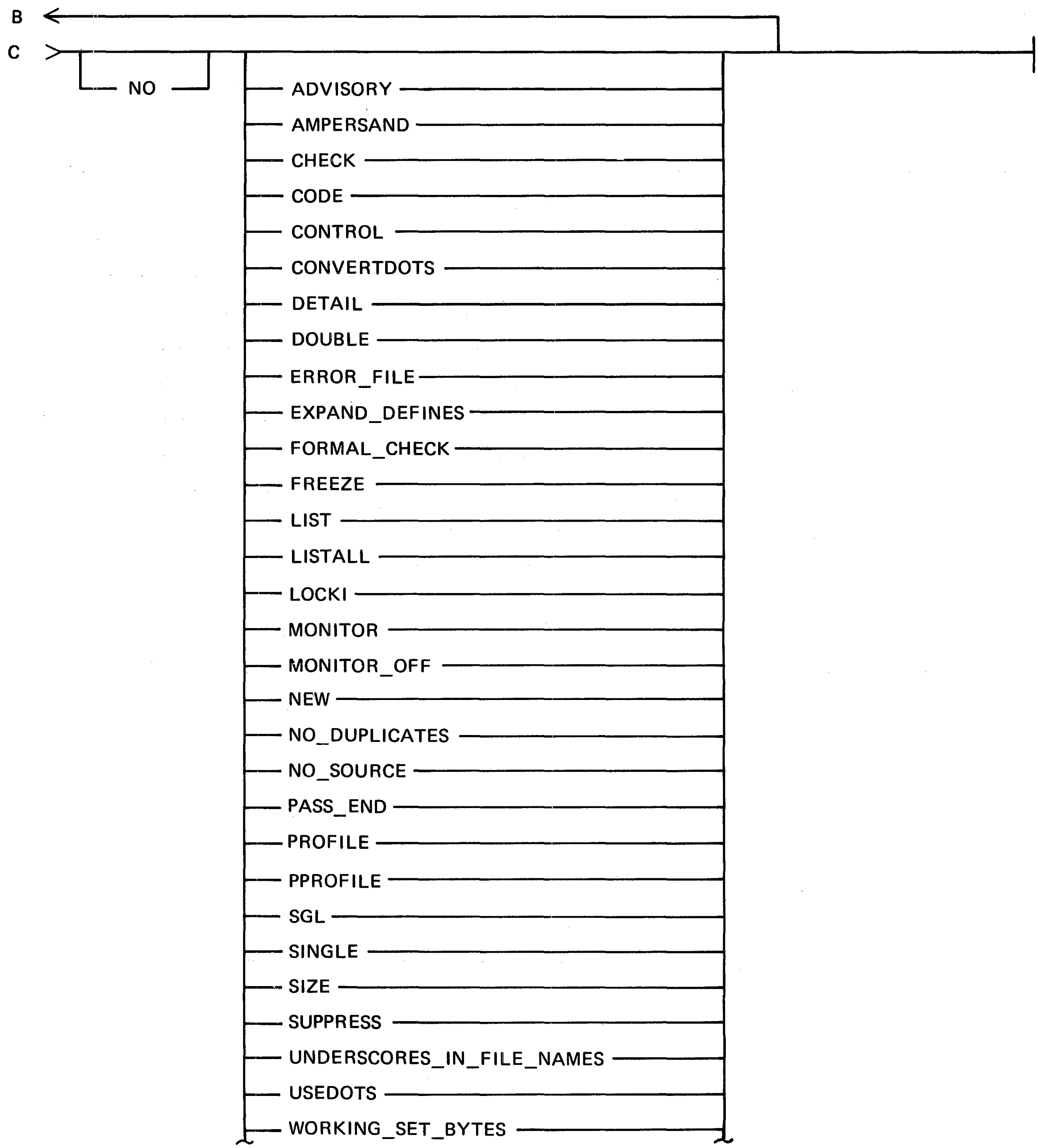
```
—— ZIP <MCP-command>; ——————————————————————————————————————————————————————|
```

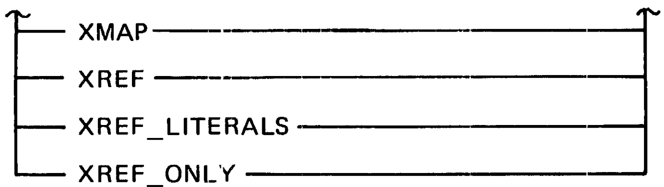
Compiler Options

The following are the syntax diagrams for the compiler options.

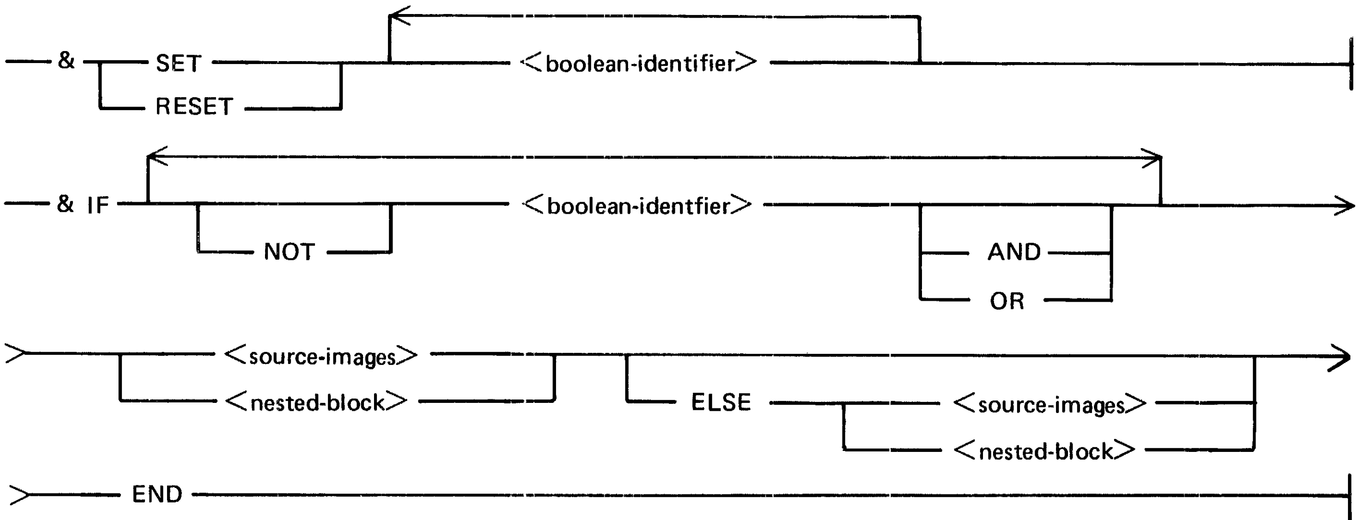
Compiler-Directing Options




B
C
NO
ADVISORY
AMPERSAND
CHECK
CODE
CONTROL
CONVERTDOTS
DETAIL
DOUBLE
ERROR_FILE
EXPAND_DEFINES
FORMAL_CHECK
FREEZE
LIST
LISTALL
LOCKI
MONITOR
MONITOR_OFF
NEW
NO_DUPLICATES
NO_SOURCE
PASS_END
PROFILE
PPROFILE
SGL
SINGLE
SIZE
SUPPRESS
UNDERSCORES_IN_FILE_NAMES
USEDOTS
WORKING_SET_BYTES



Conditional Compilation



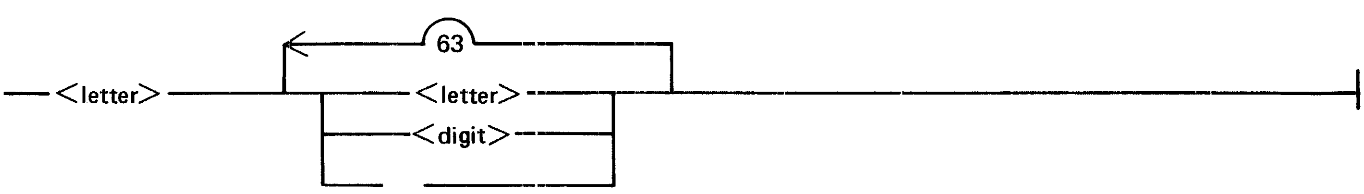
UPL RAILROAD SYNTAX GUIDE

All of the railroad syntax diagrams valid for the UPL compiler are presented in this subsection.

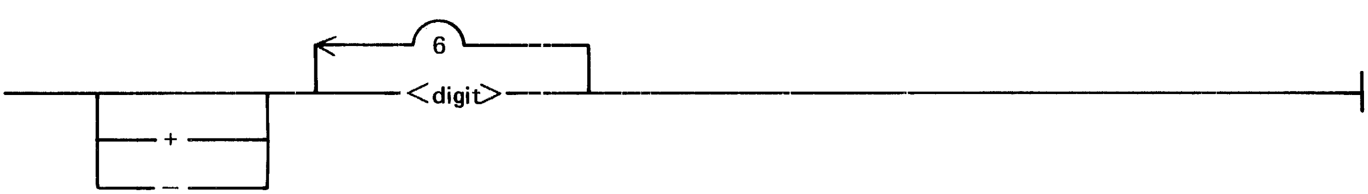
Fundamentals

The following are the syntax diagrams for the fundamental items.

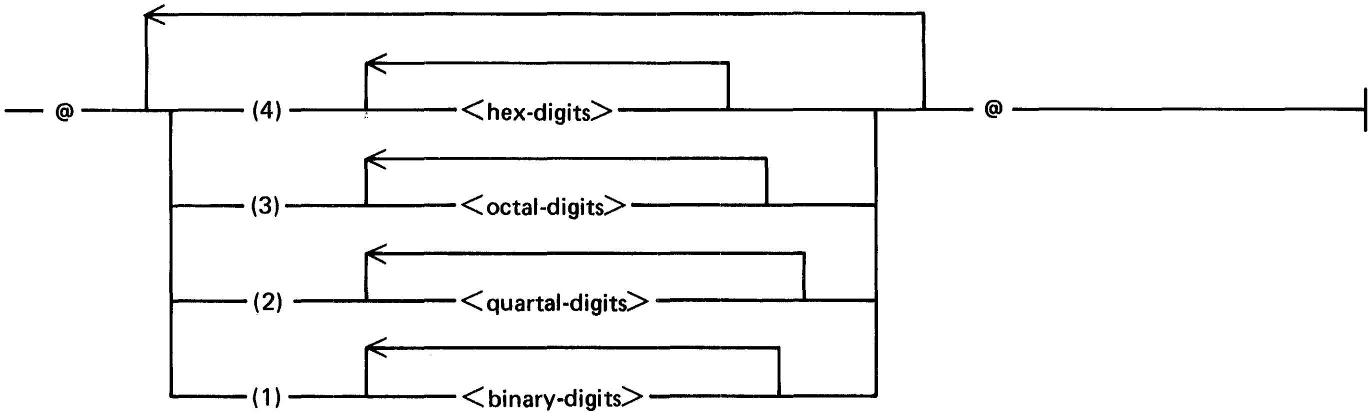
Identifiers



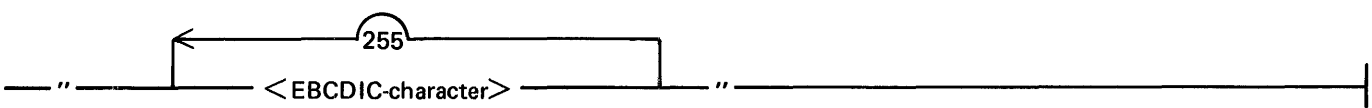
Numeric Literal



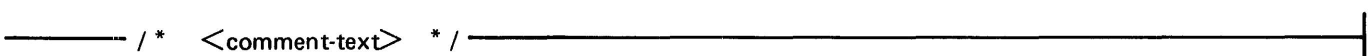
Bit-String Literal



Character-String Literal



Enclosed Comment

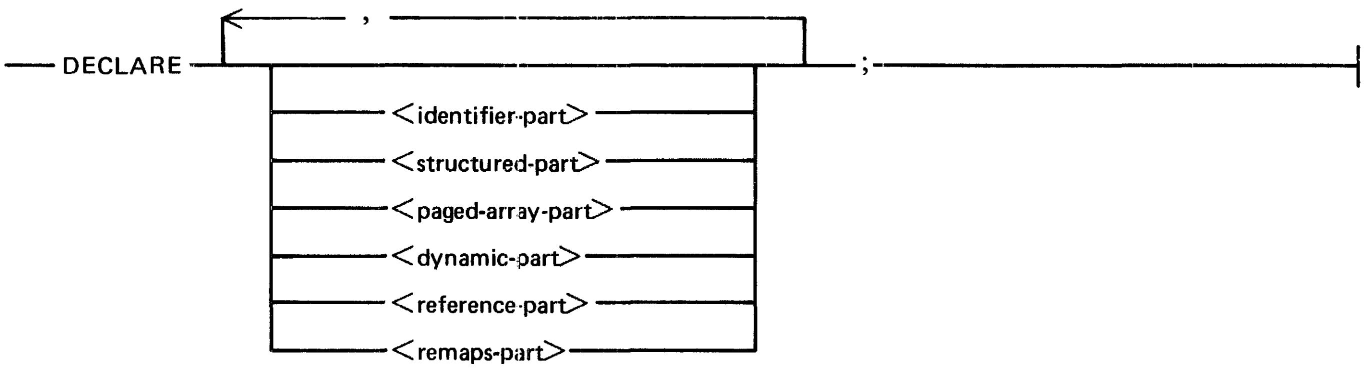


End-of-Record Comment

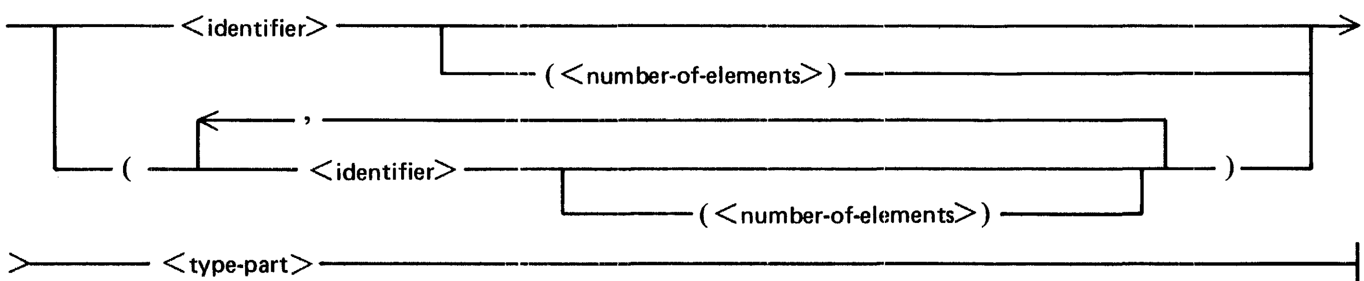
Declarations

The following are the syntax diagrams for the data, record, file, and switch-file declarations.

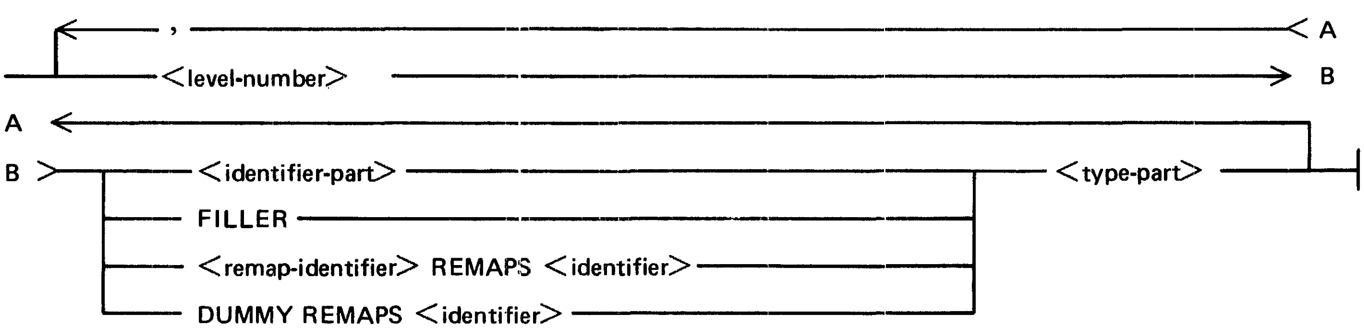
Data Declarations



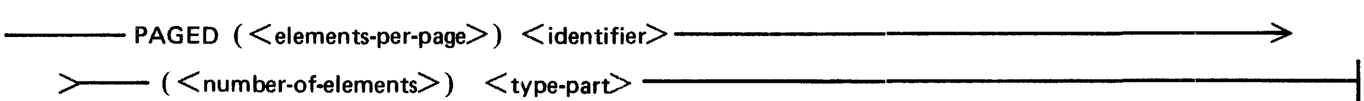
<identifier-part>



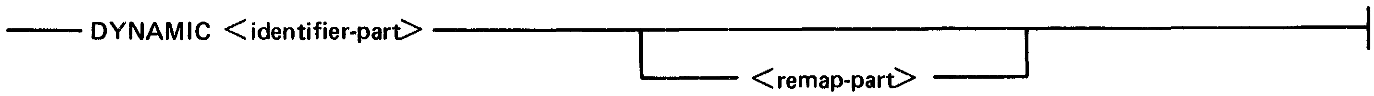
<structured-part>



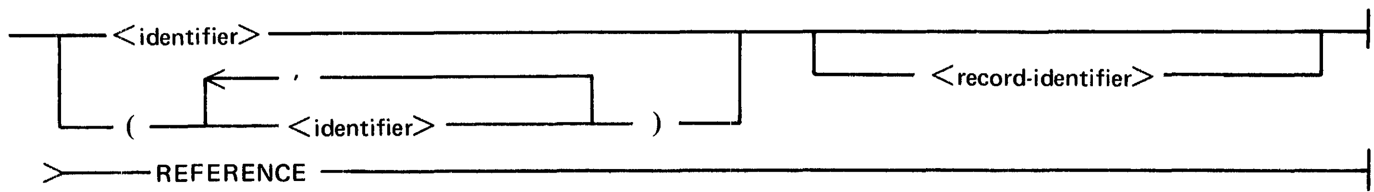
<paged-array-part>



<dynamic-part>

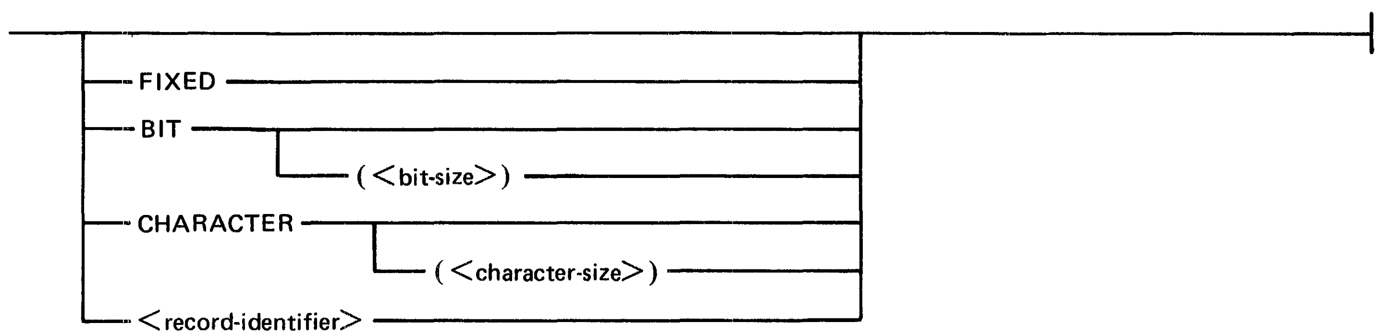


<reference-part>

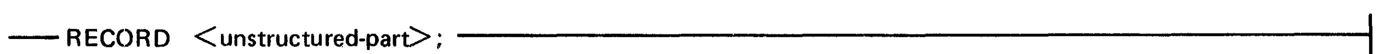


<remaps-part>

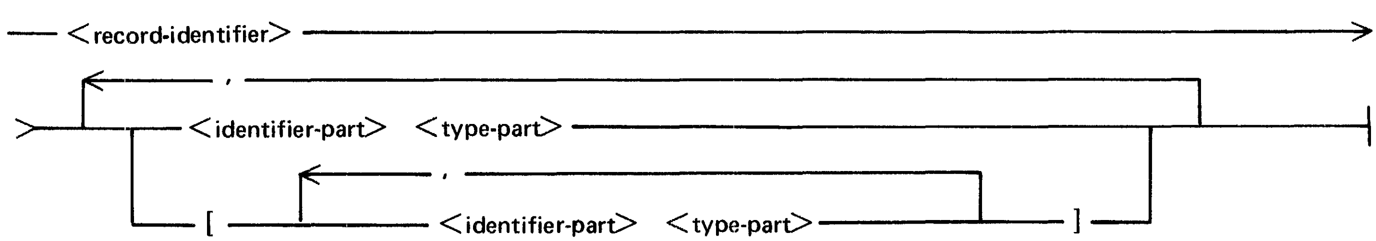
<type-part>



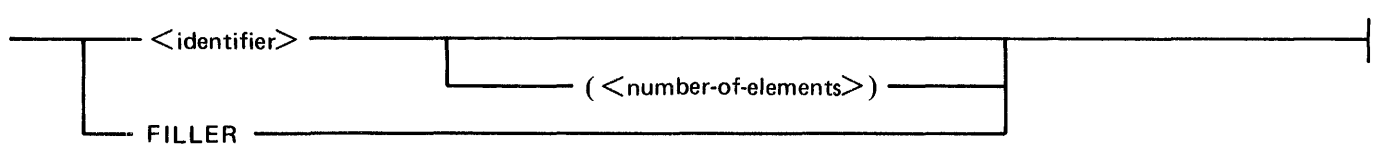
Record Declarations



<unstructured-part>

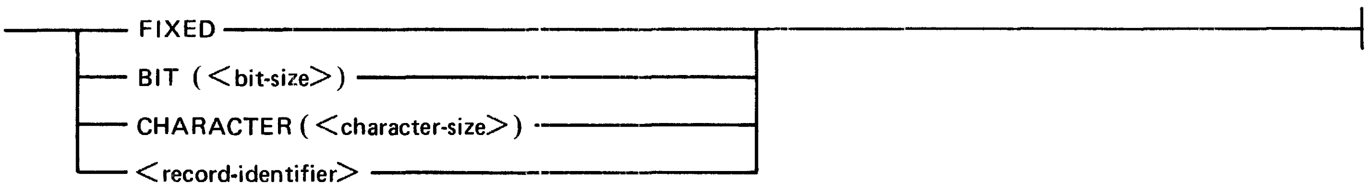


<identifier-part>

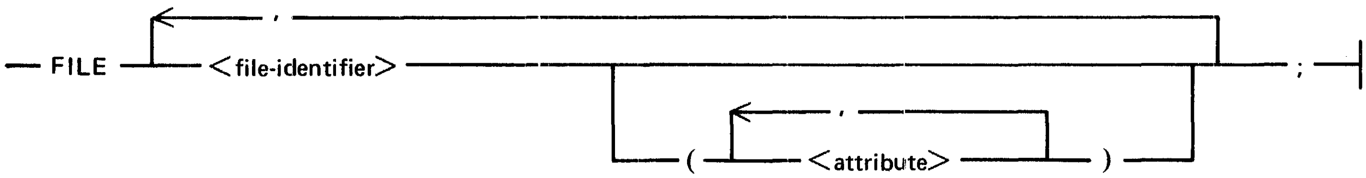


<remaps-part>

<type-part>



File Declarations

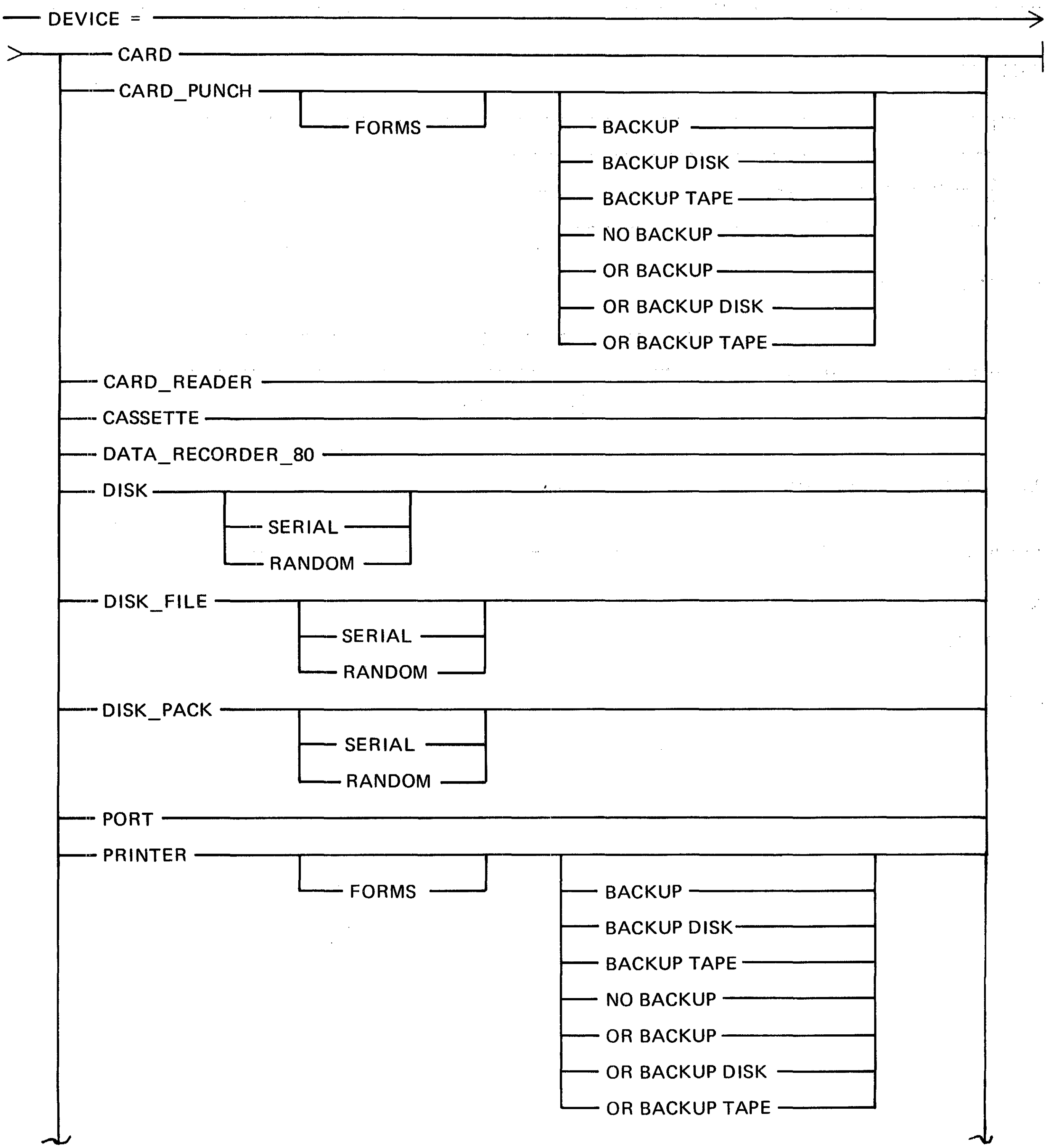


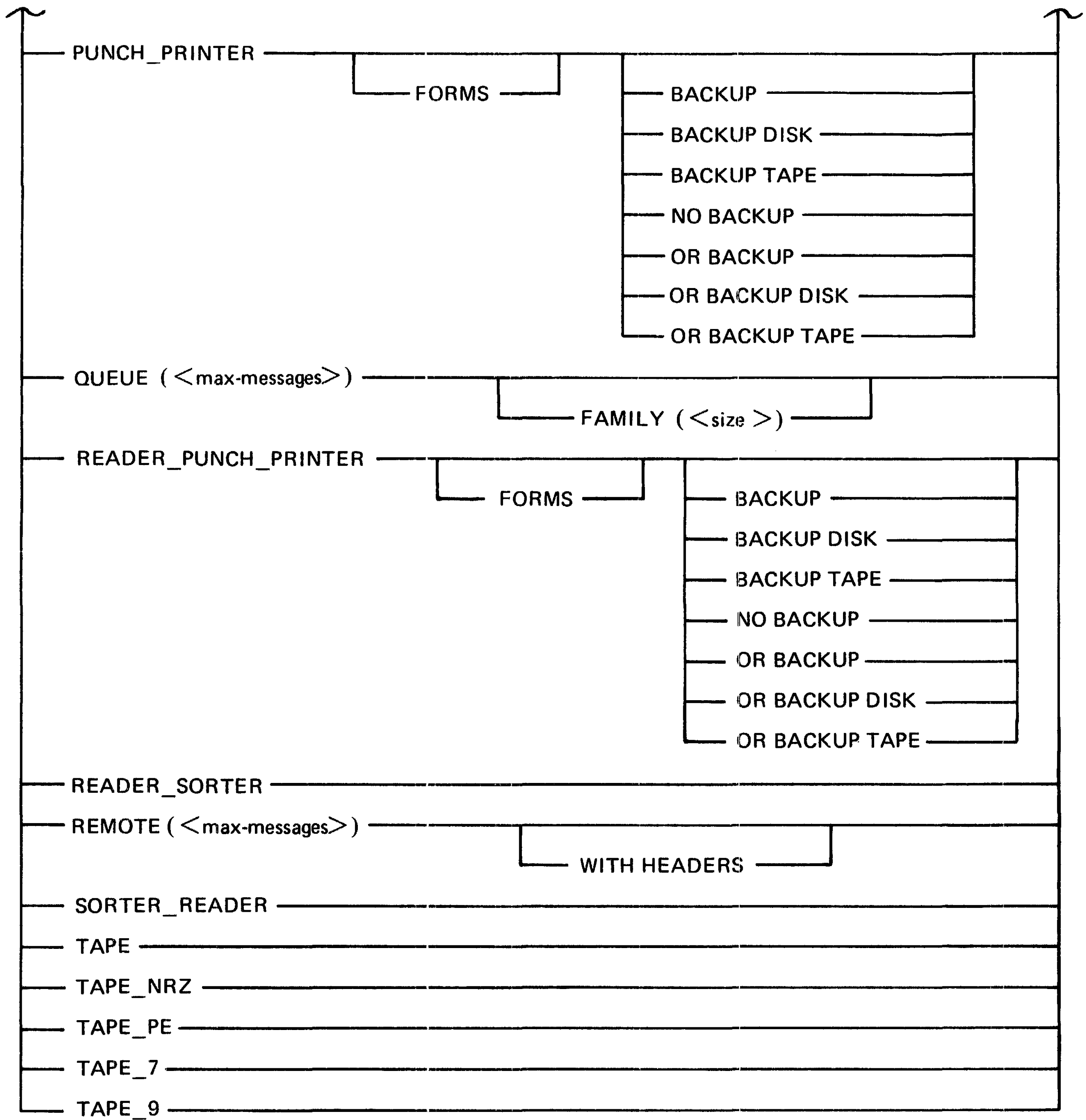
ALL_AREAS_AT_OPEN

AREAS

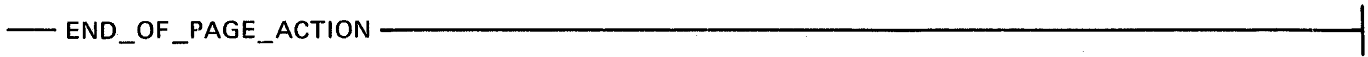
BUFFERS

DEVICE


DEVICE =
CARD
CARD_PUNCH
FORMS
BACKUP
BACKUP DISK
BACKUP TAPE
NO BACKUP
OR BACKUP
OR BACKUP DISK
OR BACKUP TAPE
CARD_READER
CASSETTE
DATA_RECORDER_80
DISK
SERIAL
RANDOM
DISK_FILE
SERIAL
RANDOM
DISK_PACK
SERIAL
RANDOM
PORT
PRINTER
FORMS
BACKUP
BACKUP DISK
BACKUP TAPE
NO BACKUP
OR BACKUP
OR BACKUP DISK
OR BACKUP TAPE


PUNCH_PRINTER
FORMS
BACKUP
BACKUP DISK
BACKUP TAPE
NO BACKUP
OR BACKUP
OR BACKUP DISK
OR BACKUP TAPE
QUEUE (<max-messages>)
FAMILY (<size>)
READER_PUNCH_PRINTER
FORMS
BACKUP
BACKUP DISK
BACKUP TAPE
NO BACKUP
OR BACKUP
OR BACKUP DISK
OR BACKUP TAPE
READER_SORTER
REMOTE (<max-messages>)
WITH HEADERS
SORTER_READER
TAPE
TAPE_NRZ
TAPE_PE
TAPE_7
TAPE_9

END_OF_PAGE_ACTION

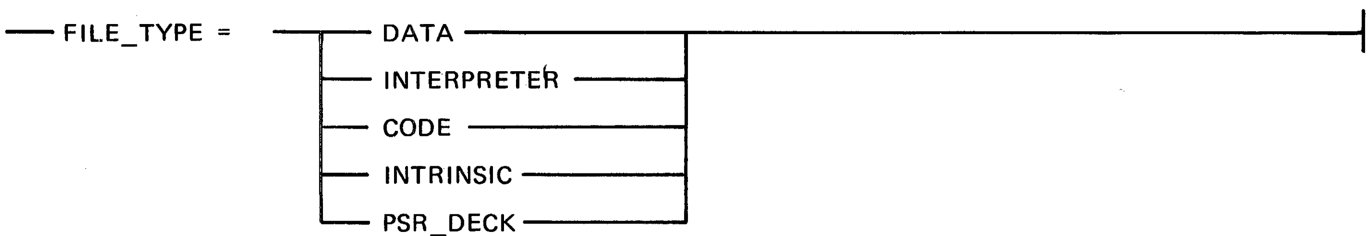


EU_INCREMENTED

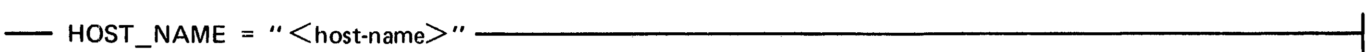
EU_SPECIAL

EXCEPTION_MASK

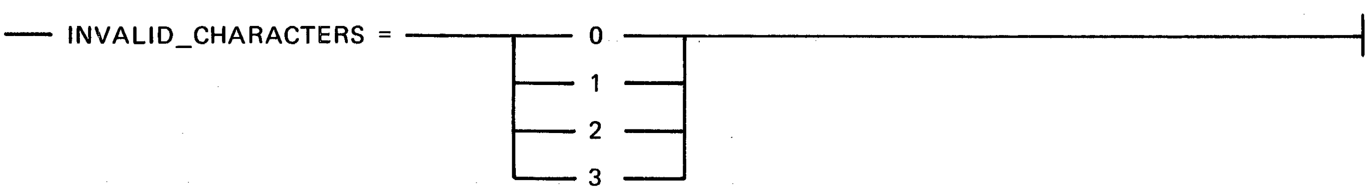
FILE_TYPE



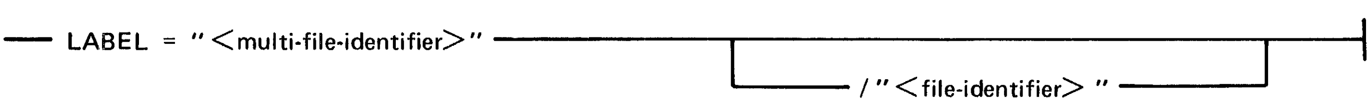
HOST_NAME



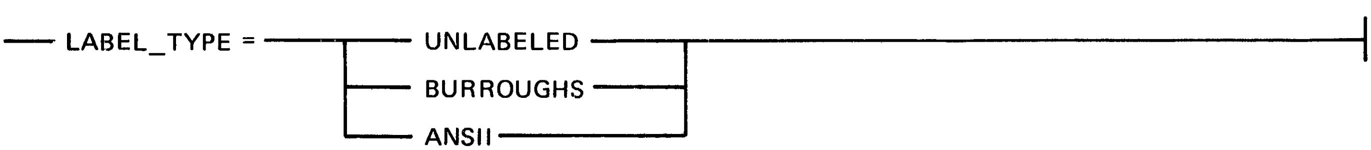
INVALID_CHARACTERS



LABEL

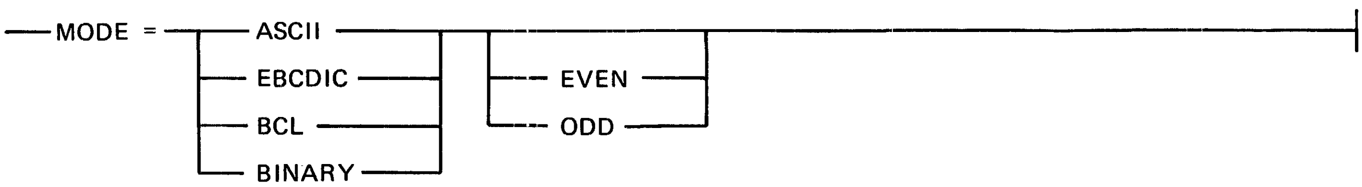


LABEL_TYPE



LOCK

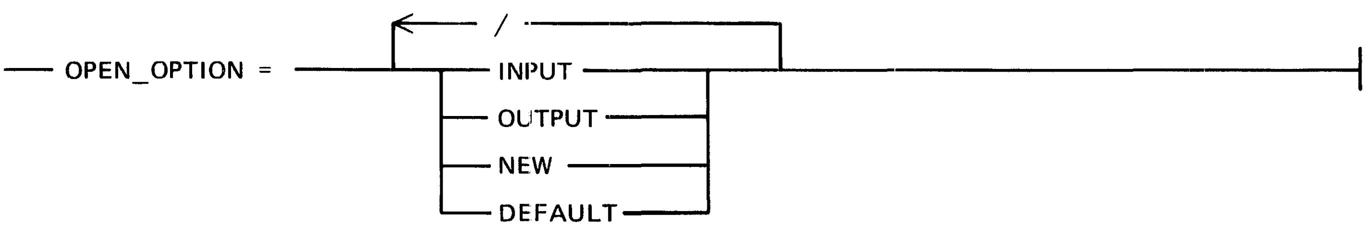
MODE



MULTI_PACK

NUMBER_OF_STATIONS

OPEN_OPTION



OPTIONAL

PACK_ID

PROTECTION

PROTECTION_IO

RECORDS

— RECORDS = ┬ <physical-size> ┬
            └ <logical-size>/<records-per-block> ┘

REEL

— REEL = <reel-number>

REMOTE_KEY

— REMOTE_KEY

SAVE

— SAVE = <number-of-days>

SECURITYTYPE

— SECURITYTYPE = <number>

SECURITYUSE

— SECURITYUSE = <number>

SERIAL

— SERIAL = ┬ <number> ┬
           └ "<character-string>" ┘

TRANSLATE

— TRANSLATE = "<file-identifier>"

USE_INPUT_BLOCKING

— USE_INPUT_BLOCKING

USER_NAMED_BACKUP

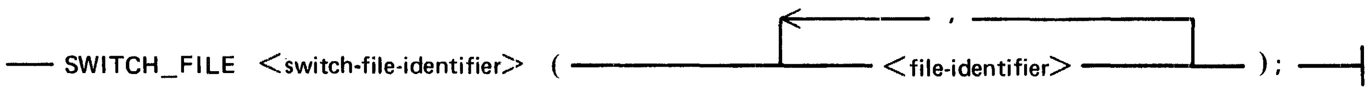
— USER_NAMED_BACKUP

VARIABLE

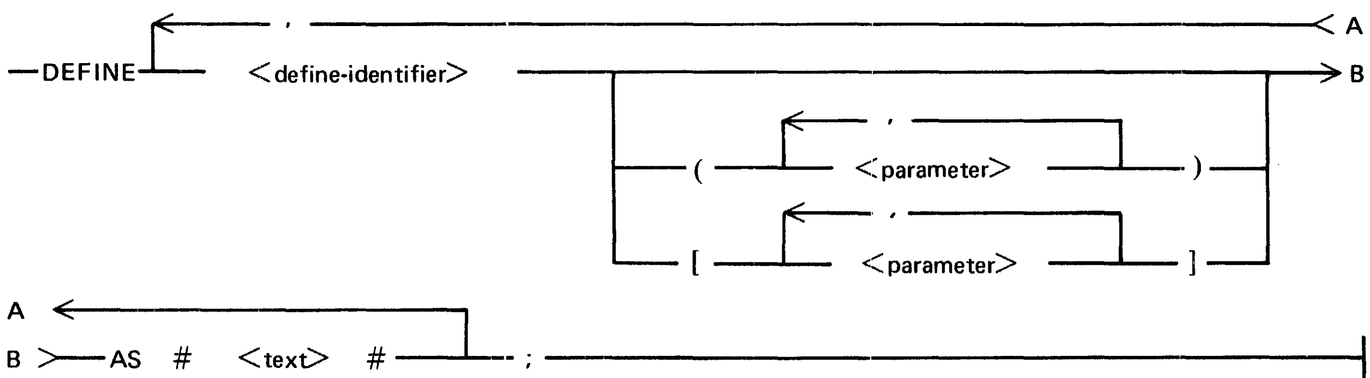
— VARIABLE

WORK_FILE

Switch File Declarations



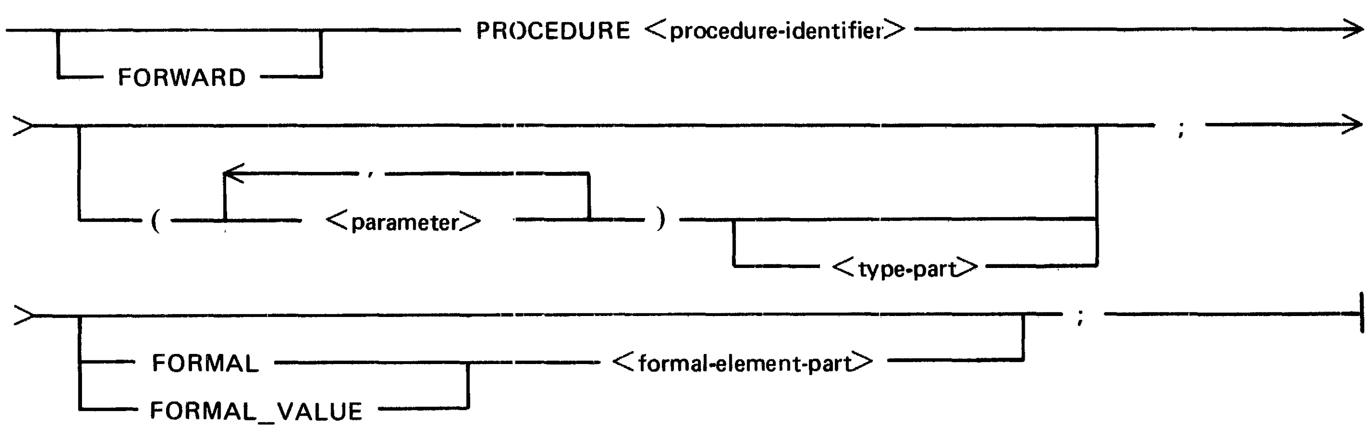
Define Statement



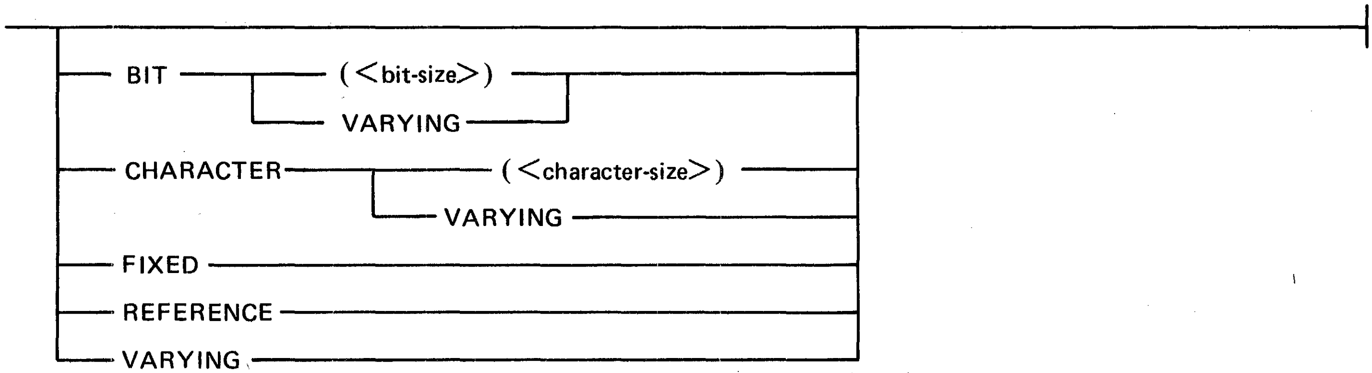
Procedure Statement

The following are the syntax diagrams for the procedure declaration, procedure body, procedure end, and procedure invocation statements.

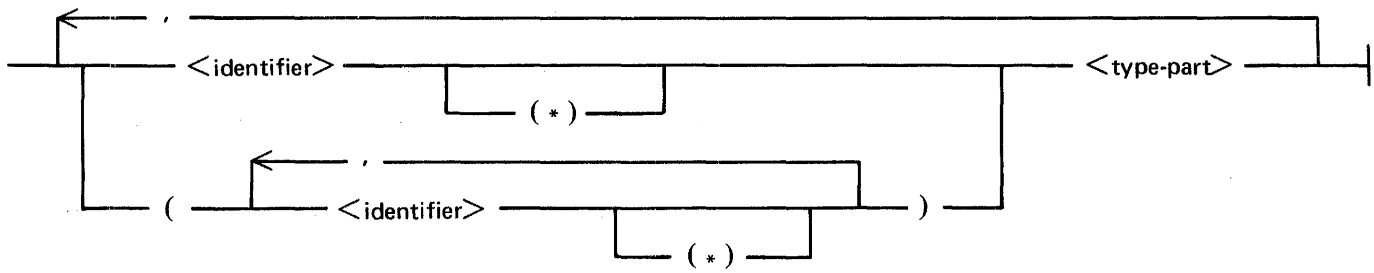
Procedure Declaration



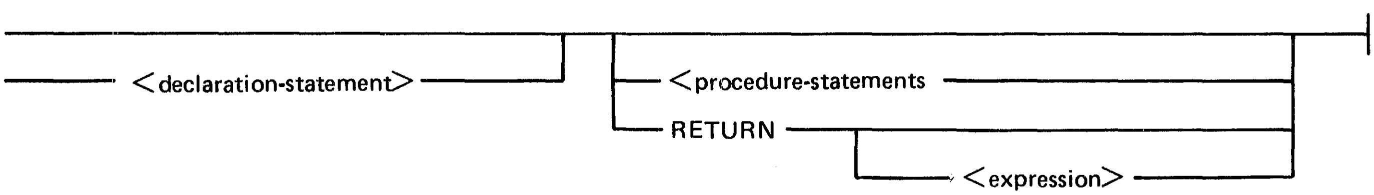
<type-part>



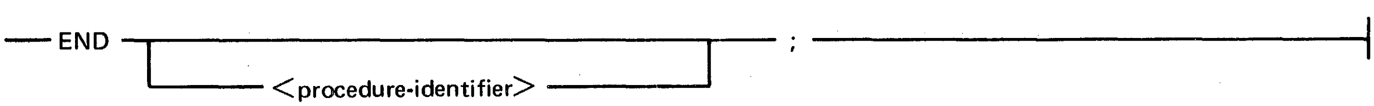
<formal-element-part>



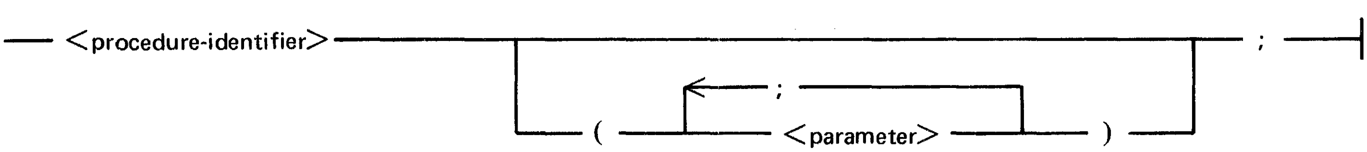
Procedure Body



Procedure End



Procedure Invocations



Expressions

The following are the syntax diagrams for the expressions.

```
──┬──────────────────────┬── <operand> ──────────────>
  └── <unary-operator> ──┘

>──┬── <operator> ──┬──────────────────────┬── <operand> ──┬──┤
   │                └── <unary-operator> ──┘               │
   └───────────────────────────────────────────────────────┘
```

Verbs

The following are the syntax diagrams for the verbs.

ACCEPT

```
── ACCEPT <destination>; ──────────────────────────────┤
```

ACCESS_FILE_INFORMATION

```
── ACCESS_FILE_INFORMATION (<file-identifier>, ──┬── BIT ────────┬──>
                                                 └── CHARACTER ──┘
>── , <destination>); ─────────────────────────────────┤
```

BINARY

```
── BINARY (<character-string>) ────────────────────────┤
```

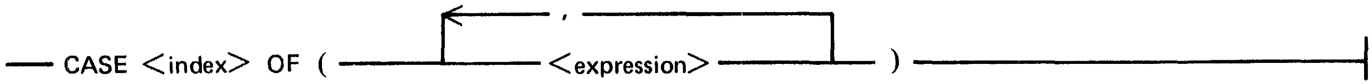
BUMP

```
── BUMP <identifier> ──┬────────────────────────────┬──┤
                       └── BY <increment-amount> ───┘
```

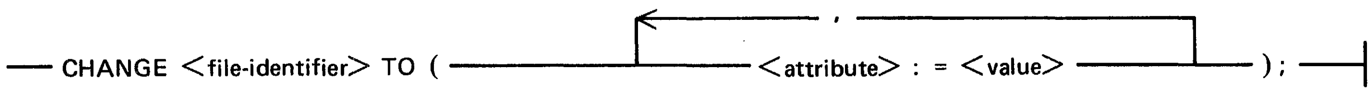
CASE (format-1)

```
── CASE <index>; ──────────────────────────────────────>
    >── <statement-0>; ────────────────────────────────>
    >── <statement-1>; ────────────────────────────────>
              .
              .
              .
    >── <statement-n>; ────────────────────────────────>
    >── END CASE; ─────────────────────────────────────┤
```

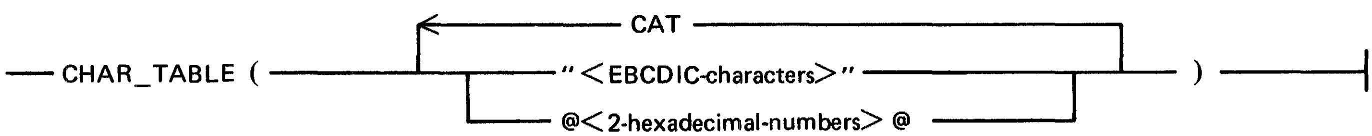
CASE (format-2)



CHANGE

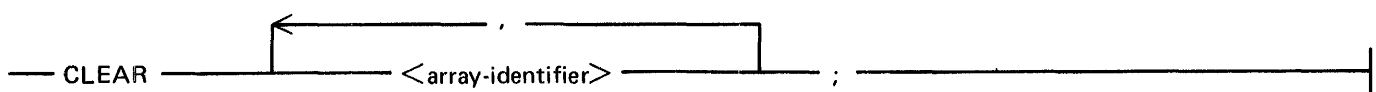


CHAR_TABLE:

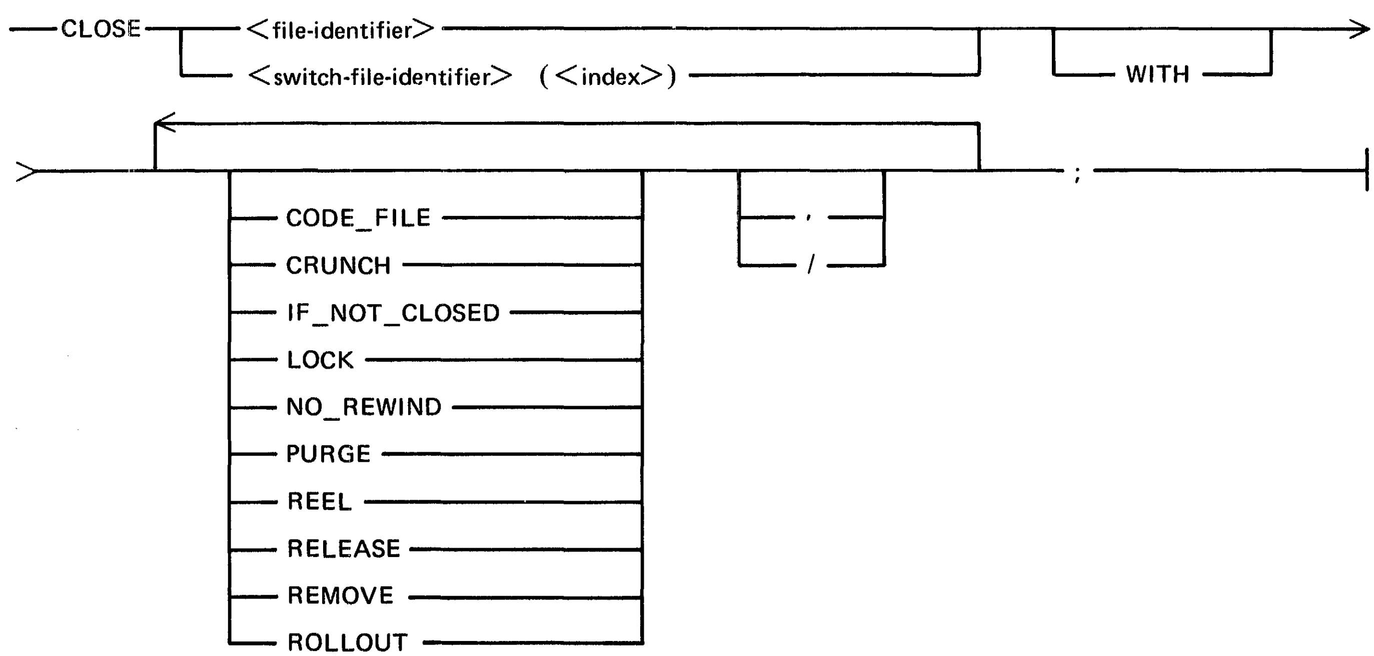


CHARACTER_FILL

CLEAR



CLOSE



COMPILE_CARD_INFO

```
── COMPILE_CARD_INFO ( <destination> ) ; ─────────────────────────────┤
```

CONSOLE_SWITCHES

```
── CONSOLE_SWITCHES ──────────────────────────────────────────────────┤
```

CONVERT

```
── CONVERT ( <convert-value>, ──┬── FIXED ───────────────────────┬── ) ──┤
                                ├── BIT ──────┬──┐               │
                                └── CHARACTER ┘  └── , ──┬── 4 ──┤
                                                         ├── 3 ──┤
                                                         ├── 2 ──┤
                                                         └── 1 ──┘
```

DATA_ADDRESS

```
── DATA_ADDRESS ( <identifier> ) ─────────────────────────────────────┤
```

DATE

```
── DATE ─┬────────────────────────────────────────────────────────┬──┤
         └── ( ──┬── DAY ──────┬── , ──┬── BIT ────────┬── ) ─────┘
                 ├── JULIAN ───┤       ├── CHARACTER ──┤
                 ├── MONTH ────┤       └── DIGIT ──────┘
                 └── YEAR ─────┘
```

DECIMAL

```
── DECIMAL ( <string>, <string-size> ) ───────────────────────────────┤
```

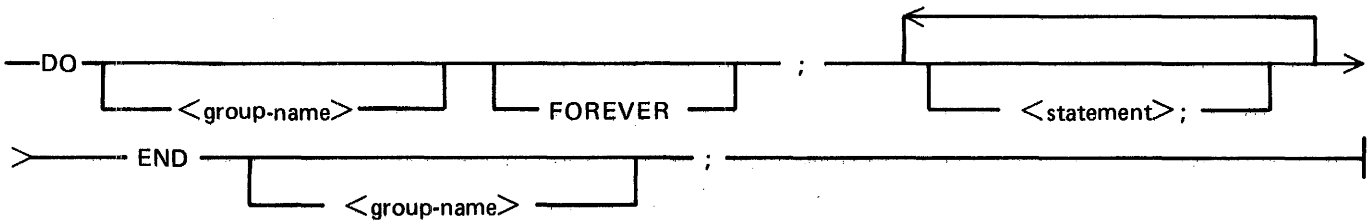
DECREMENT

```
── DECREMENT < identifier> ─────────┬──────────────────────────────┬──┤
                                    └── BY < decrement-amount> ────┘
```

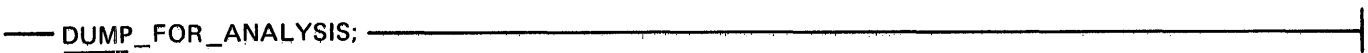
DISPLAY

```
── DISPLAY ( <display-identifier> ) ───┬──────────────────┬── ; ──┤
                                       └── , CRUNCHED ────┘
```

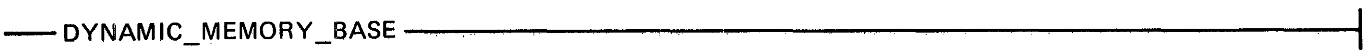
DO



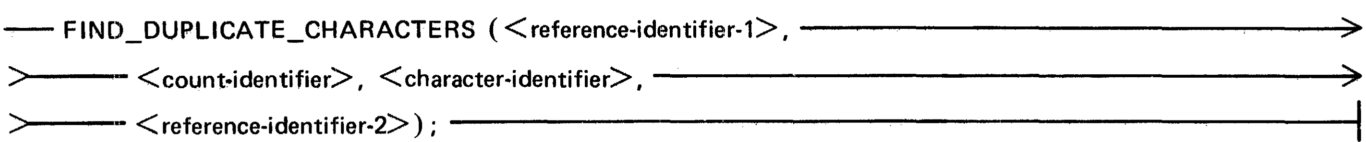
DUMP_FOR_ANALYSIS



DYNAMIC_MEMORY_BASE



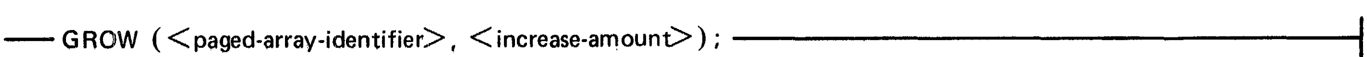
FIND_DUPLICATE_CHARACTERS



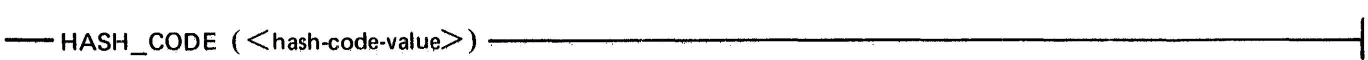
FINI

FREEZE_PROGRAM

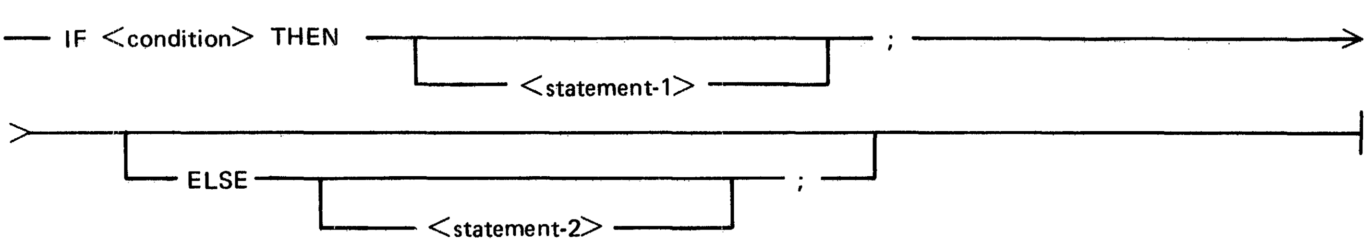
GROW



HASH_CODE



IF, THEN, and ELSE (Conditionals)



LENGTH

```
—— LENGTH (<identifier>) ——————————————————————————————————|
```

LIMIT_REGISTER

```
—— LIMIT_REGISTER ——————————————————————————————————————————|
```

LOCATION

```
—— LOCATION ( ——————————┬— <identifier> ——————————————┬— ) ————|
                        ├— <array-identifier> ————————┤
                        └— <procedure-identifier> ————┘
```

MAKE_READ_ONLY

```
—— MAKE_READ_ONLY (<paged-array-identifier>), <page-number>); ——————|
```

MAKE_READ_WRITE

```
—— MAKE_READ_WRITE (<paged-array-identifier>, <page-number>), ——————|
```

MESSAGE_COUNT

```
—— MESSAGE_COUNT (<queue-file-id>, <identifier>); ————————————————|
```

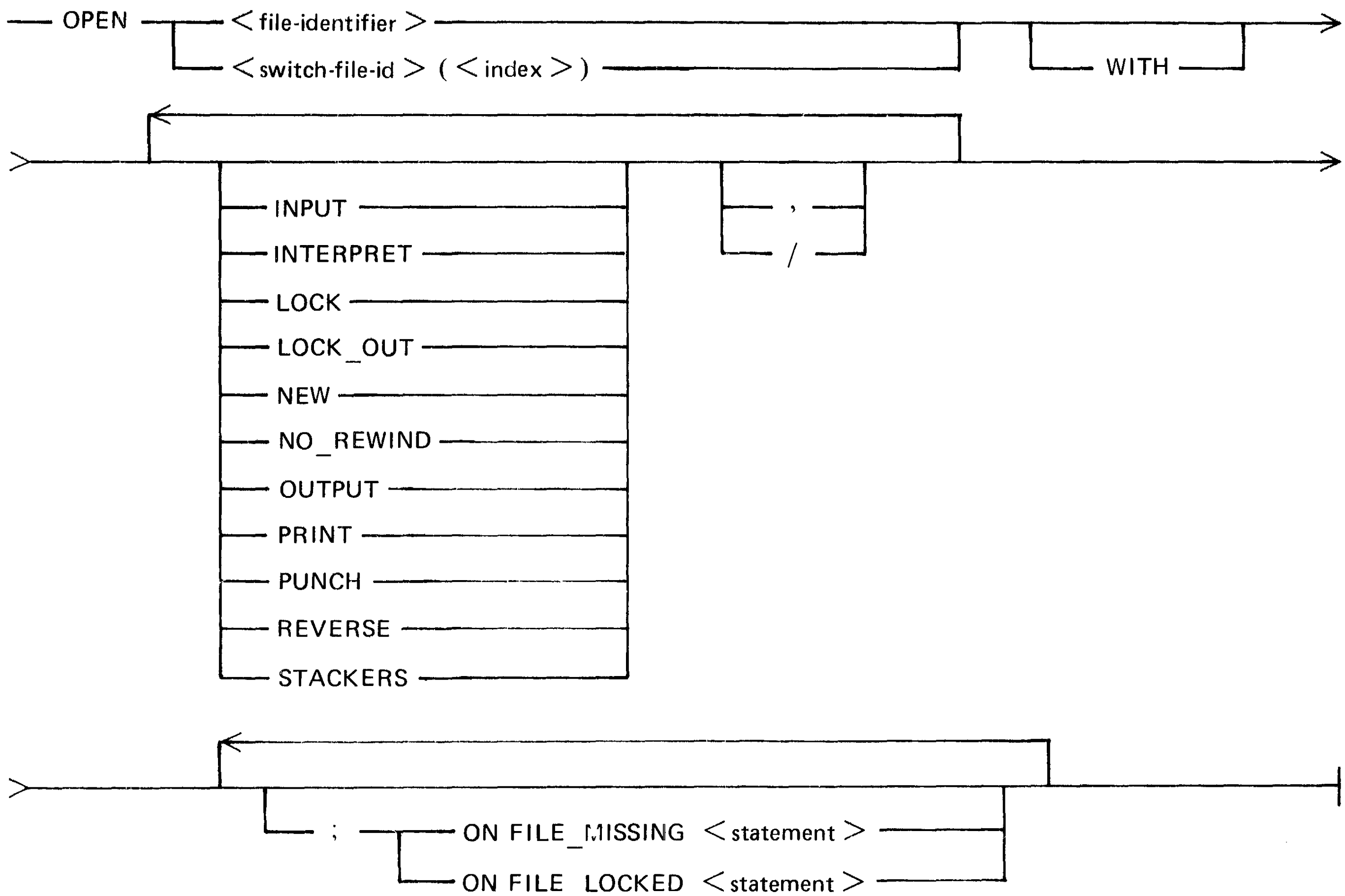
M_MEM_SIZE

```
—— M_MEM_SIZE ——————————————————————————————————————————————|
```

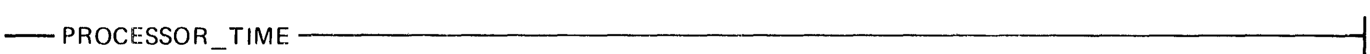
NAME_OF_DAY

```
—— NAME_OF_DAY —————————————————————————————————————————————|
```

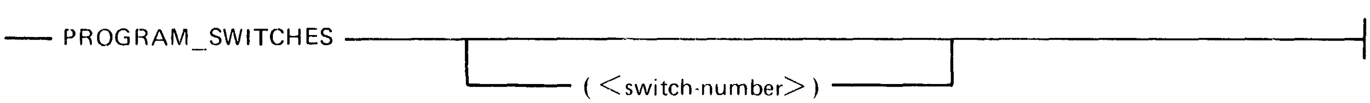
OPEN



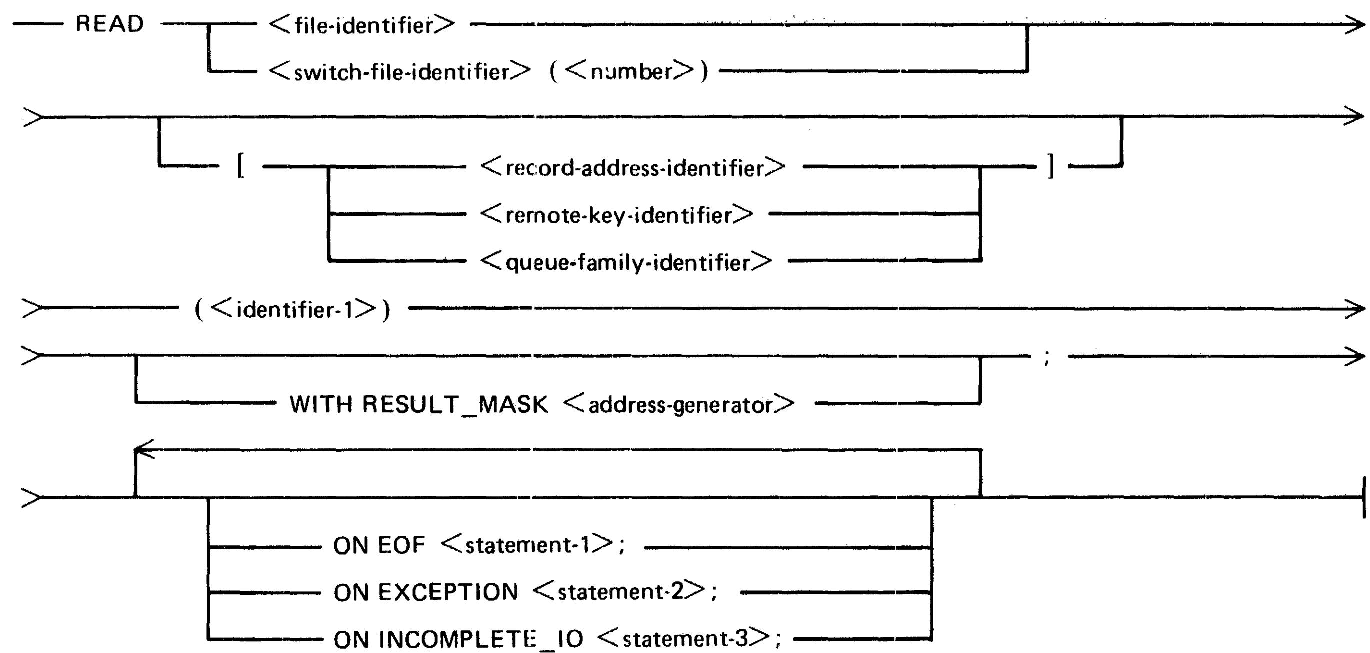
PROCESSOR_TIME



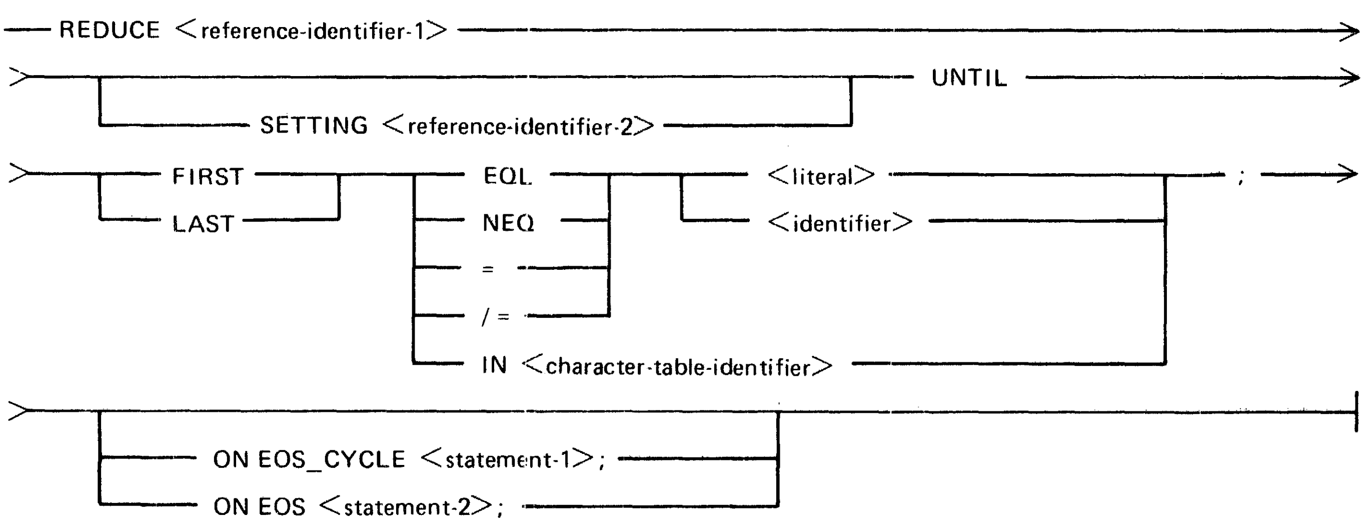
PROGRAM_SWITCHES



READ



REDUCE



REFER

RETURN

— RETURN ;
<expression>

REVERSE_STORE

— REVERSE_STORE (<address-generator-1>, <address-generator-2>,
> < > , <address-generator-n>, <expression>)

SEARCH_DIRECTORY

— SEARCH_DIRECTORY (<file-identifier, <identifier>,
> BIT);
CHARACTER
ON FILE_MISSING <statement-1>;
>
ON FILE_LOCKED <statement-2>;

SEARCH_LINKED_LIST

— SEARCH_LINKED_LIST (<first-item>, <compare-field>,
> <compare-value>, <relation>, <link-field>);

SEEK

— SEEK <file-identifier> [<record-address-identifier>] ;

SEGMENT_PAGE

— SEGMENT_PAGE (<segment-identifier>
1
> , IMPORTANT);
OF <page-identifier>

SKIP

— SKIP <file-identifier> TO <channel-number>;

SORT

```
— SORT ( <sort-information-table>, <key-table>, ———>
>——— <input-file-identifier>, <output-file-identifier> ———>
>————————————————————————————— ); ——————|
   └——— , <translate-file-identifier> ——┘
```

SORT_MERGE

```
— SORT_MERGE ( <sort-information-table>, <key-table>, ———>
>——— <merge-input-table>, <output-file-identifier> ———>
>————————————————————————————— ); ——————|
   └——— , <translate-file-identifier> ——┘
```

SORT_SWAP

```
— SORT_SWAP ( <identifier-1>, <identifier-2> ) ; ——————|
```

SPACE

```
— SPACE <file-identifier> ——┬——————┬—— <space-amount> ——┬—— ; ———>
                            ├— TO —┘                    │
                            └— TO_EOF ——————————————————┘

>——┬————————————————————————————————————————┬——————|
   │ ┌<———————————————————————————————————┐ │
   └—┴┬— 1 —— ON EOF <statement-1> ; ————┬┘
      └— 1 —— ON EXCEPTION <statement-2> ; ┘
```

SPO_INPUT_PRESENT

```
— SPO_INPUT_PRESENT ——————————————————|
```

STOP

```
— STOP ——┬————————————————————┬—— ; ——————|
         └— <syntax-errors> ——┘
```

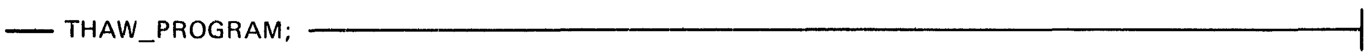
SUBBIT

```
— SUBBIT ( <string-identifier>, <start-position> ——┬——————————————┬—— ) ——|
                                                   └— , <length> —┘
```

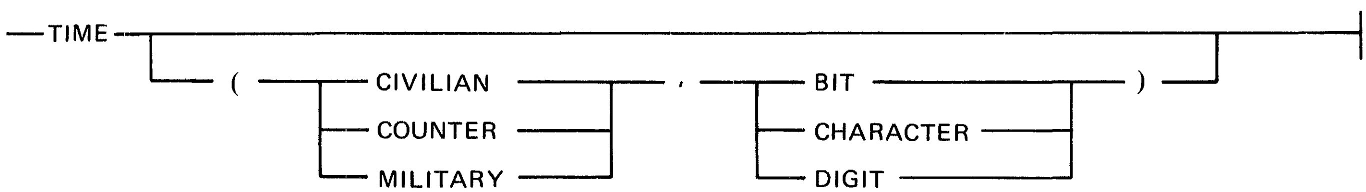
SUBSTR

SWAP

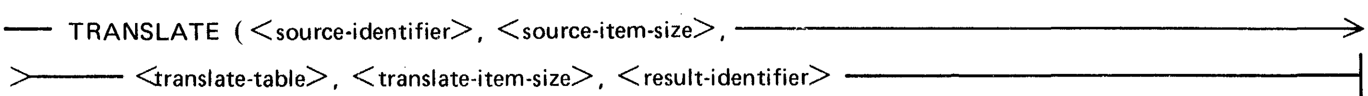
THAW__PROGRAM



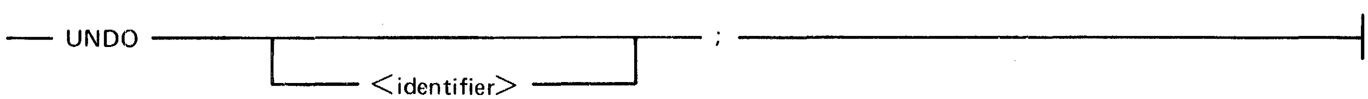
TIME



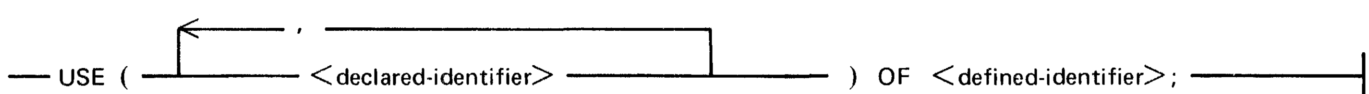
TRANSLATE



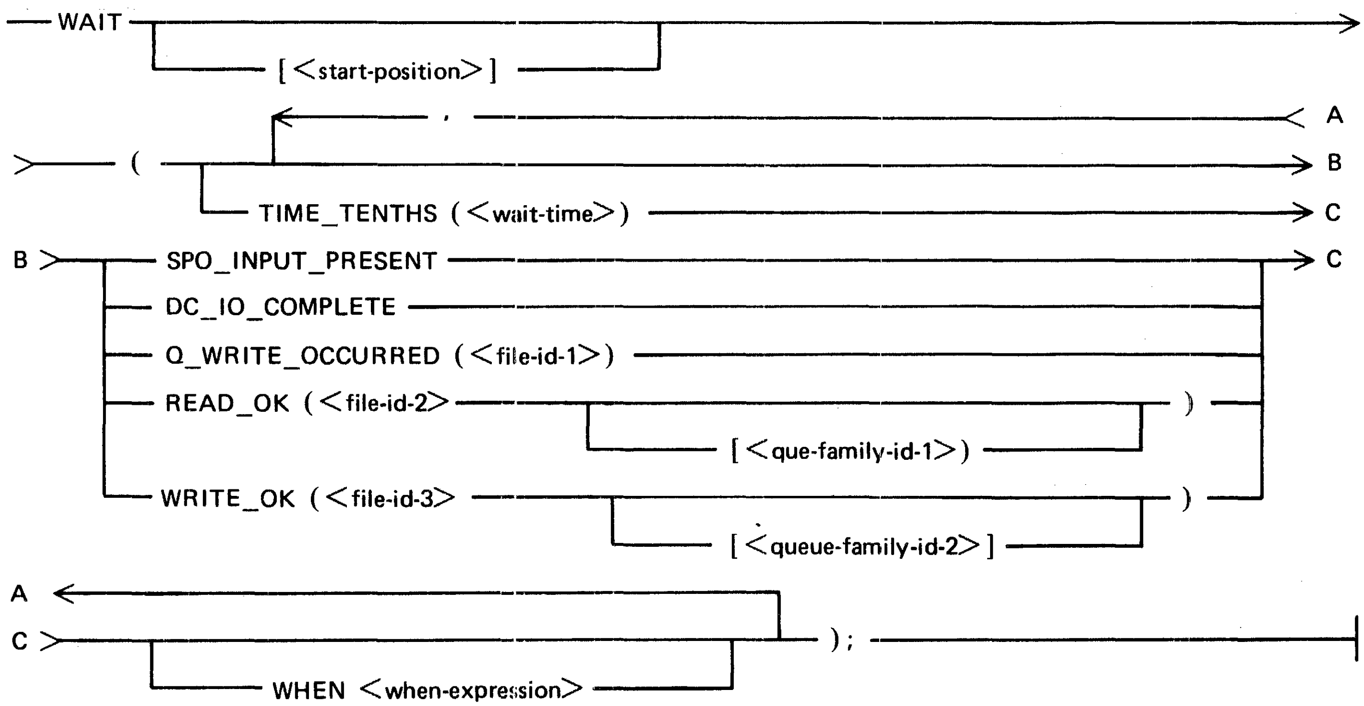
UNDO



USE



WAIT


WAIT
[<start-position>]
,
A
(
B
TIME_TENTHS (<wait-time>)
C
B
SPO_INPUT_PRESENT
C
DC_IO_COMPLETE
Q_WRITE_OCCURRED (<file-id-1>)
READ_OK (<file-id-2>
)
[<que-family-id-1>)
WRITE_OK (<file-id-3>
)
[<queue-family-id-2>]
A
C
);
WHEN <when-expression>

WRITE

```
—— WRITE ——┬────────────┬──┬── <file-identifier> ─────────────────────────────┬──>
           └── LOCK ────┘  └── <switch-file-id> (<identifier-1>) ────────────┘

>──┬──────────────────────────────────────────────────────────────┬──>
   ├── DOUBLE ────────────────────────────────────────────────────┤
   ├── NEXT ──────────────────────────────────────────────────────┤
   ├── NO ────────────────────────────────────────────────────────┤
   ├── PAGE ──────────────────────────────────────────────────────┤
   ├── SINGLE ────────────────────────────────────────────────────┤
   ├── <channel-number> ──────────────────────────────────────────┤
   └── [ ──┬── <record-address-identifier> ──┬── ] ───────────────┘
           ├── <remote-key-identifier> ──────┤
           └── <queue-family-identifier> ────┘

>──┬──────────┬── (<record>) ─────────────────────────────────────>
   └── TOP ───┘

>──┬────────────────────────────────────────────────┬── ; ────────>
   └── WITH RESULT_MASK <address-generator> ────────┘

>──┬──────────────────────────────┬───────────────────────────────>
   └── ON EOF <statement-1>; ─────┘

>──┬──────────────────────────────────────┬───────────────────────>
   └── ON EXCEPTION <statement-2> ; ──────┘

>──┬──────────────────────────────────────────┬──────────────────┤
   └── ON INCOMPLETE_IO <statement-3>; ───────┘
```

X_ADD

```
—— X_ADD (<expression-1>, <expression-2>) ──────────────────────┤
```

X_DIV

```
—— X_DIV (<expression-1>, <expression-2> ) ─────────────────────┤
```

X_MOD

```
—— X_MOD (<expression-1>, <expression-2>) ──────────────────────┤
```

X_MUL

```
—— X_MUL (<expression-1>, <expression-2>) ──────────────────────┤
```

X_SUB

```
── X_SUB (<expression-1>, <expression-2>) ──────────────────────────┤
```

ZIP

```
── ZIP <MCP-command>; ───────────────────────────────────────────────┤
```

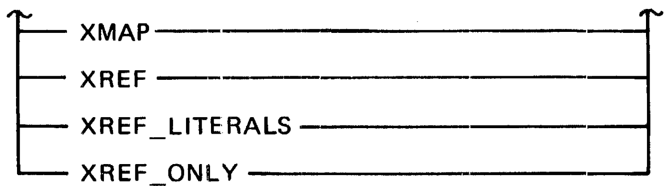
Compiler Options

The following are the syntax diagrams for the compiler options.

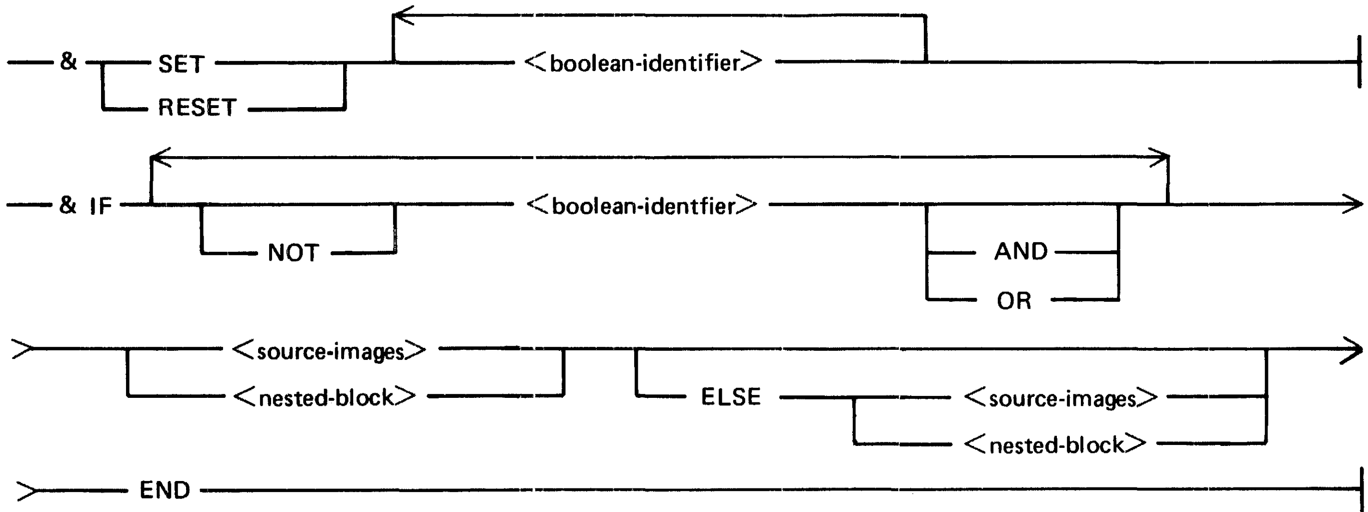
Compiler-Directing Options

```
──┬─ $ ─┬─┬──────────────────┬─┬─ RECOMPILE ──┬───────────────────────┬─┬──> A
  └─ & ─┘ └─ CREATE_MASTER ──┘ │              └─ "<multi-file-id>" ───┘ │
                               └────────────────────────────────────────┘

          ┌<──────────────────────────────────────────────────────────< B
A >───────┴───┬──────────────────────────────────────────────────┬───> C
              ├─ CSSIZE <cssize-number> ─────────────────────────┤
              ├─ DEBUG <sequence-number> ────────────────────────┤
              ├─ DYNAMICSIZE <dynamicsize-number> ───────────────┤
              ├─ ESSIZE <essize-number> ─────────────────────────┤
              ├─ INTERPRETER "<file-identifier>" ────────────────┤
              ├─ INTRINSIC "<multi-file-identifier>" ────────────┤
              ├─ LIBRARY "<file-identifier>" ────────────────────┤
              ├─ LIBRARY_PACK "<pack-identifier>" ───────────────┤
              ├─ MERGE ──────────────────────────────────────────┤
              ├─ NSSIZE <nssize-number> ─────────────────────────┤
              ├─ PAGE ───────────────────────────────────────────┤
              ├─ PPSSIZE <ppssize-number> ───────────────────────┤
              ├─ RECOMPILE_TIMES ────────────────────────────────┤
              ├─ SEQ <base-sequence-number> ──┬──────────────────┤
              │                               └─ + <increment> ──┤
              ├─ VOID ──┬───────────────────────────────┬────────┤
              │         └─ <ending-sequence-number> ────┘        │
              └─ VSSIZE <vssize-number> ─────────────────────────┘
```

B
C
NO
ADVISORY
AMPERSAND
CHECK
CODE
CONTROL
CONVERTDOTS
DETAIL
DOUBLE
ERROR_FILE
EXPAND_DEFINES
FORMAL_CHECK
FREEZE
LIST
LISTALL
LOCKI
MONITOR
MONITOR_OFF
NEW
NO_DUPLICATES
NO_SOURCE
PASS_END
PROFILE
PPROFILE
SGL
SINGLE
SIZE
SUPPRESS
UNDERSCORES_IN_FILE_NAMES
USEDOTS
WORKING_SET_BYTES



Conditional Compilation



APPENDIX D
GLOSSARY OF COMMONLY USED TERMS AND ACRONYMS

absolute address

1. An address that identifies a storage location or a device without the use of any intermediate reference.
2. An address that is permanently assigned by the machine designer to a storage facility.

address

1. A character or group of characters that identifies a register, a particular part of storage, or some other data source or destination.
2. To refer to a device or an item of data by its address.

address part
A part of an instruction that usually contains only an address or part of an address.

address register
A register in which an address is stored.

algorithm
A finite set of well-defined rules for the solution of a problem in a finite number of steps.

alphabet
An ordered set of all the letters used in a language, but does not include punctuation marks.

alphabetic character set
A character set that contains letters and may contain control characters, special characters, and the space character, but not digits.

alphanumeric
Pertaining to a character set that contains letters, digits, and usually other characters such as punctuation marks.

alphanumeric character set
A character set that contains both letters and digits and may contain control characters, special characters, and the space character.

alphanumeric data
Data represented by letters and digits, perhaps with special characters and the space character.

American Standard Code for Information Interchange (ASCII)
The standard code, using a coded character set of 7-bit coded characters (8-bits including parity check), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

AND
A logic operator having the property that if P is a statement, Q is a statement, R is a statement, ..., then the AND of P, Q, R, ... is TRUE if all statements are TRUE, FALSE if any statement is FALSE.

application program
A user-written program that performs tasks.

arithmetic instruction
An instruction in which the operation part specifies an arithmetic operation.

arithmetic operation
An operation that follows the rules of arithmetic.

array
An arrangement of elements in one or more dimensions.

ASCII
The acronym for American Standard Code for Information Interchange.

assignment statement
An instruction used to express a sequence of operations, or used to assign operands to specified variables, symbols, or both.

base address

1. A numeric value that is used as a reference in the calculation of addresses in the execution of a computer program.
2. A given address from which an absolute address is derived by combination with a relative address.

beginning of job (BOJ)
The execution of a single program unit to be performed by the system.

binary
Pertaining to a selection, choice, or condition that has two possible different values or states.

binary arithmetic operation
An arithmetic operation in which the operands and the result are represented in the pure binary system.

binary code
A code that makes use of only two distinct characters, usually 0 and 1.

binary digit (bit)
In binary notation, either of the characters 0 or 1.

binary search
A search in which, at each step of the search, the set of items is partitioned into two equal parts, some appropriate action being taken in the case of an odd number of items.

bit
In the pure binary system, either the digit 0 and 1. Synonymous with binary digit.

bit string
A string consisting solely of bits.

blank
A part of a data medium in which no characters are recorded.

blank character
A graphic representation of the space character.

block length

1. The number of records, words, or characters in a block.
2. A measure of the size of a block, usually specified in units such as records, words, computer words, or characters.

BOJ
The acronym for beginning of job.

boolean
Pertaining to the processes used in the algebra formulated by George Boole.

boolean operation

1. An operation in which each of the operands and the result take one of two values.
2. An operation that follows the rules of boolean algebra.

boolean operator
An operator in which each of the operands and the result take one of two values.

buffer
A storage area used to compensate for a difference in rate of flow of data, or in time of occurrence of events, when transferring data from one device to another.

buffer storage
A storage device that is used to compensate for differences in the rate of flow of data between components or, within an automatic data processing system, for the time of occurrence of events in the components.

byte
A binary character string operated upon as a unit and usually shorter than a computer word.

call

1. The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point.
2. In data communication, the action performed by the calling party, or the operations necessary in making a call, or the effective use of a connection between two stations.
3. To transfer control to a specified closed subroutine.

card image
A one-to-one representation of the hole patterns of a punched card.

carriage control tape

1. A tape that is used to control vertical tabulation of printing positions or display positions.
2. A tape that contains line feed control data for a printing device.

central processing unit (CPU)
A unit of a computer that includes circuits that control the interpretation and execution of instructions.

channel

1. A path along which signals can be sent: for example, data channel, output channel.
2. In data communication, a means of one-way transmission.

character
A digit, letter, or special character.

character set
An agreed upon, finite set of unique characters.

clear

1. To put one or more storage locations or registers into a prescribed state, usually that denoting 0 (zero).
2. To cause one or more storage locations to be in a prescribed state, usually corresponding to 0 (zero) or corresponding to the space character.

comment
A description, reference, or explanation added to or interspersed among the statements of the source language. Comments do not affect program execution.

compare
To examine two items to determine their relative magnitudes, their relative positions in an order or sequence, or to determine whether they are identical in given characteristics.

compile

1. To translate a computer program expressed in a problem-oriented language into a computer-oriented language.
2. To prepare a machine language program from a computer program written in another programming language by: 1) making use of the overall logic structure of the program, or 2) generating more than one computer instruction for each symbolic statement, or a combination of (1) and (2), and (3) performing the function of an assembler.

compiler
A computer program used to compile.

complement
A number that can be derived from a specified number by subtracting it from a second specified number.

computer instruction
An instruction that can be recognized by the central processing unit of the computer for which it is designed.

computer language
A language in which the instructions consist only of computer instructions.

computer-oriented language
A programming language that reflects the structure of a particular computer or class of computers. A programming language in which the words and syntax are designed for use on a specific class of computers.

computer program
A program expressed in a form suitable for execution by a computer.

constant
See figurative constant.

control character
A character occurring in a particular context to initiate, modify, or stop a control operation. A control character may be recorded for use in a subsequent action. A control character is not a graphic character, but may have a graphic representation in some circumstances.

control operation
An action that affects the recording, processing, transmission, or interpretation of data: for example, starting or stopping a process, carriage return, rewind, and end of transmission.

control state
A term used to refer to a program that can assume control of the system's processor with privileged operands. The type of control state program suggested here usually means an operating system or MCP.

convert
To change the representation of data from one form to another without changing the information they convey.

CPU
The acronym for central processing unit.

cycle
An interval of space or time in which one set of events or phenomena is completed. Any set of operations that is repeated regularly in the same sequence. The operations may be subjected to variations on each repetition.

data

1. A representation of facts, concepts, or instructions in a formalized manner suitable for communication, interpretation, or processing manually or automatically.
2. Any representations such as characters or analog quantities to which meaning can be assigned.

data attribute
A characteristic of a unit of data such as length, value, or method of representation.

data base
A set of data, the whole or part of another set of data, and consisting of at least one file that is sufficient for a given purpose or for a given data processing system.

data type
Declares the identifier as BIT, CHARACTER, or FIXED.

debug
To detect, trace, and eliminate mistakes in computer programs or other software.

decimal

1. Pertaining to a selection, choice, or condition that has ten possible values or states.
2. Pertaining to a number system having ten digit places.

decimal digit
In decimal notation, or in the decimal number system, one of the digits 0 to 9.

decimal notation
A notation that uses ten different characters, usually the decimal digits.

declaration
In a programming language, a meaningful expression that affects the interpretation of other expressions in that language.

declare statement
A statement that names a variable and assigns a memory location and data attributes to that name.

default option
An implicit option that is assumed when no option is explicitly stated.

define
Assigns a section of source code to an identifier.

delimiter
A flag that separates and organizes items of data.

difference
In a subtraction operation, the number or quantity that is the result of subtracting the subtrahend from the minuend.

digit
A graphic character that represents an integer: for example, one of the characters 0 to 9.

directory
A table of identifiers and references to the corresponding items of data.

disk cartridge
A secondary data storage device much the same as a disk pack and usually smaller in size. It can be moved on line or off line.

disk directory
A disk-resident table that contains the name and type of file, together with a pointer to the disk file header or subdirectory for all permanent files which reside on the disk.

disk pack
A removeable assembly of magnetic disks. A portable set of flat, recording surfaces used in a disk storage device.

display
A visual presentation of data.

display device
An output unit that gives a visual representation of data. Usually the data are displayed temporarily; however, arrangements may be made for making a permanent record.

dividend
In a division operation, the number or quantity to be divided.

divisor
: In a division operation, the number or quantity by which the dividend is divided.

EBCDIC
: The acronym for Extended Binary Coded Decimal Interchange Code

element
: In a set, an object, entity, or concept having the properties that define a set.

end of job (EOJ)
: The termination of a single program unit to be performed by the system.

EOJ
: The acronym for end of job.

error
: A discrepancy between a computed, observed, or measured value or condition and the TRUE, specified, or theoretically correct value or condition.

error message
: An indication that an error has been detected.

exclusion
: The 2-operand boolean operation whose result has the boolean value 1 if the first operand has the boolean value 1 and the second has the boolean value 0.

exclusive-OR
: A logic operator having the property that if P is a statement and Q is a statement, then P exclusive-OR Q is TRUE if either but not both statements are TRUE, FALSE if both are TRUE or both are FALSE.

exclusive-OR element
: A logic element that performs the boolean nonequivalence operation.

execute
: In programming, to change the state of a computer in accordance with the rules of the operations it recognizes. To perform the execution of an instruction or of a computer program.

execution
: The process by which a computer program or subroutine changes the state of a computer in accordance with the rules of the operations that a computer recognizes. The process of carrying out an instruction by a computer. The process of carrying out the instructions of a computer program by a computer.

expression
: The operational portion of a program statement that produces a value.

Extended Binary Coded Decimal Interchange Code (EBCDIC)
: A coded character set consisting of 8-bit coded characters used to represent unique letters, numbers, and special characters.

factor
: In a multiplication operation, any of the numbers or quantities that are the operands.

family name
    An identifier used as a file name, or the name assigned to identify a main file with subdirectory entries. Same as multifile-id.

fetch
    To locate and load a quantity of data from storage.

field
    In a record, a specified area used for a particular category of data; for example, a group of positions in which a wage rate is recorded.

FIFO (first-in-first-out)
    A queuing technique in which the text item to be retrieved is the item that has remained in the queue the longest.

figurative constant
    A data name that is reserved for a specified constant in a specified programming language.

file
    A set of related records treated as a unit; for example, in stock control, a file could consist of a set of invoices. .ne 10

file identifier (file-id)
    All disk file identifiers used on the system must be unique to prevent duplicate file names. A file identifier can be composed of any combination of the following file identifier options:

file-id
multifile-id/file-id
disk-id/multifile-id/file-id

file maintenance
    The activity of keeping a file up to date by adding, changing, or deleting records.

filler
    One or more characters adjacent to an item of data that serve to bring its representation up to a specified size.

file security
    The procedures or special devices used to prevent access to or use of data or programs without authorization.

fixed storage
    A storage device whose contents are inherently nonerasable, nonerasable by a particular user, or nonerasable when operating under particular conditions.

flag

1. Any of various types of indicators used for identification; for example, a word mark.
2. A character that signals the occurrence of some condition, such as the end of a word.

format
    The arrangement or layout of data in or on a data medium.

generate
: To produce a computer program by a selection of subsets from skeletal code under the controls of parameters.

generator
: A controlling routine that performs a generating function; for example, report generator, I/O generator.

global
: Pertaining to that which is defined in one subdivision of a computer program, and then used in at least one other subdivision of that computer program.

graphic
: A symbol produced by a process such as handwriting, drawing, or printing.

graphic character
: A character, other than a control character, which is normally represented by a graphic.

hardware
: Physical equipment used in data processing, as opposed to computer program, procedures, rules, and associated documentation.

hash total
: The result obtained by applying an algorithm to a set of data for checking purposes; for example, a summation obtained by treating data items as numbers.

heading
: In ASCII and data communication, a sequence of characters preceded by the start-of-heading character used as machine sensible address or routing information.

high-level language
: A programming language that does not reflect the structure of any one given computer or that of any given class of computers.

identifier
: A character or group of characters used to identify or name an item of data and possibly to indicate certain properties of that data.

inclusive-OR element
: A logic element that performs the boolean operation of disjunction.

index

1. In programming, a subscript of integer value that identifies the position of an item of data with respect to some other item of data.
2. A list of the contents of a file or of a document, together with keys or references for locating the contents.
3. A symbol or numeral used to identify a particular quantity in an array of similar quantities.

indexed address
: An address that is modified by the content of an index register prior to or during the execution of a computer instruction.

index register
A register whose contents may be used to modify an operand address during the execution of computer instructions, so as to operate as a clock or counter. An index register may be used to control the execution of a loop, to control the use of an array, as a switch, for table lookup, as a pointer, etc.

initialize
To set counters, switches, addresses, or contents of storage to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.

input

1. One, or a sequence of, input states.
2. Pertaining to a device, process, or channel involved in an input process, or to the data or states involved in an input process.

input area
An area of storage reserved for input.

input data
Data being received or to be received into a device or computer program.

input-output (I/O)
Pertaining to a device or to a channel that may be involved in an input process and, at a different time, in an output process.

input unit
A device in a data processing system by which data may be entered into the system.

instruction
In a programming language, a meaningful expression that specifies one operation and identifies its operands, if any.

instruction address register
A register from whose contents the address of the next instruction is derived. An instruction address register may also be a portion of a storage device specifically designated for the derivation of the address of the next instruction by a translator, compiler, interpreter, language processor, operating system, and so forth.

instruction control unit
In central processing unit, the part that receives instructions in proper sequence, interprets each instruction, and applies the proper signal to the arithmetic and logic unit and other parts in accordance with this interpretation.

instruction counter
A counter that indicates the location of the next computer instruction to be interpreted.

instruction format
The layout of an instruction showing its constituent parts.

instruction register
A register that is used to hold an instruction for interpretation.

instruction set
The set of instructions of a computer, of a programming language, or the programming languages in a programming system.

integer
One of the numbers 0, +1, −1, +2, −2, and so forth.

internal storage
A storage device directly controlled by the central processing unit of a digital computer.

interpret
To translate and to execute each source language statement of a computer program before translating and executing the next statement.

interpreter
A computer program used to interpret.

interrupt
To stop a process in such a way that it can be resumed.

interruption
A suspension of a process, such as the execution of a computer program, normally caused by an event external to that process, and performed in such a way that it can be resumed.

I/O
The acronym for input/output.

item
One member of a group. A file may consist of a number of items, such as records, which in turn may consist of other items. A collection of related characters treated as a unit.

job
A set of data that completely defines a unit of work for a computer. A job usually includes all necessary computer programs, linkages, files, and instructions to the operating system.

justify

1. To control the printing positions of characters on a page so that both the left-hand and right-hand margins of the printing are regular.
2. To shift the contents of a register, if necessary, so that the character at a specified end of the data that has been read or loaded into the register is at a specified position in the register.
3. To align characters horizontally or vertically to fit the positioning constraints of a required format.

K
When referring to storage capacity, two to the tenth power (1024).

key
One or more characters, within a set of data, that contains information about the set, including its identification.

keypunch
A keyboard-actuated device that punches holes in a punch card or a punched card.

keyword
: One of the predefined words of an artificial language.

label
: One or more characters, within or attached to a set of datà, that contains information about the set, including its identification.

language
: A set of characters, conventions, and rules, that are used for conveying information. The three aspects of language are pragmatics, semantics, and syntax.

language processor
: A computer program that performs such functions as translating and interpreting and other tasks required for processing a specified programming language.

leading zero
: In positional notation, a zero in a more significant digit place than the digit place of the significant nonzero digit of a numeral. .ne 8

left-justify
: To shift the contents of a register so that the data is moved to a specified position. To control the printing positions of characters on a page so that the left-hand margin of the printing is regular.

letter
: A graphic character which when used alone or combined with others, represents in a written language one or more sound elements of a spoken language, but excludes marks used alone and punctuation.

level
: The degree of subordination of an item in a hierarchic arrangement.

level number
: A reference number that indicates the position of an item in a hierarchic arrangement.

lexicographic level
: A lexicographic (lexic) level is a compile-time relationship of each procedure to the outer level of the program. The outer level is referred to as level 0 (zero). All other procedures are nested within lexic level 0 and are assigned a lexic level number representing their depth of nesting from lexic level 0.

library
: A collection of related files. For example, one line of an invoice may form an item, a complete invoice may form a file, the collection of inventory control files may form a library, and the libraries used by an organization are known as its data bank.

library routine
: A computer program in or from a program library.

LIFO (last-in-first-out)
: A queuing technique in which the next item to be retrieved is the item most recently placed in the queue.

line printer
: A device that prints a line of characters as a unit.

link
In computer programming, the part of a computer program, in some cases a single instruction or an address, that passes control and parameters between separate portions of the computer program.

list
An ordered set of items of data.

literal
In a source program, an explicit representation of the item value which must be unaltered during any translation.

load
In computer programming, to enter data into storage or working registers.

local
Pertaining to that which is defined and used only in one subdivision of a computer program.

location
Any place in which data may be stored.

logical record
A record independent of its physical environment. Portions of the same logical record may be located in different physical records, or several logical records or parts of logical records may be located in one physical record.

loop
A set of instructions that may be executed repeatedly while a certain condition prevails. In some implementations, no test is made to discover whether the condition prevails until the loop has been executed once.

machine language
A language that is used directly by a machine.

machine-readable medium
A medium that can convey data to a given sensing device.

mask
A pattern of characters used to control the retention or elimination of portions of another pattern of characters. To use a pattern of characters to control the retention or elimination of portions of another pattern of characters.

master file
A file which is used as an authority in a given job and which is relatively permanent, even though its contents may change.

memory
See main storage.

merge
To combine the items of two or more sets that are each in the same given order into one set in that order.

minuend
In subtraction, the number or quantity from which another number or quantity is subtracted.

mnemonic symbol
A symbol chosen to assist the human memory; for example, an abbreviation such as "mpy" for "multiply".

module
A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading: for example, the input to and output from, an assembler, compiler, or executive routine.

modulo-n counter
A counter in which the number represented reverts to zero in the sequence of counting after reaching a maximum value of n − 1.

multifile-id
See family name.

multiplicand
In a multiplication operation, the factor that is multiplied by another number or quantity.

multiplier
In multiplication, the number or quantity by which the multiplicand is multiplied.

multiprocessing
A mode of operating a multiprocessor that provides for the parallel processing of two or more computer programs. Pertaining to the simultaneous execution of two or more computer programs or sequences of instructions by a computer or computer network.

multiprocessor
A computer employing two or more central processing units under integrated control.

multiprogramming
A mode of operation that provides for the interleaved execution of two or more computer programs by a single central processing unit. Pertaining to the concurrent execution of two or more computer programs by a computer.

n-ary
Pertaining to a selection, choice, or condition that has n possible different values or states.

negate
To perform the operation of negation.

negation
A boolean operation the result of which has the boolean value opposite to that of the operand.

nest
To embed procedures or DO-groups into other procedures or DO-groups at a different hierarchical level such that the different levels can be performed or accessed recursively.

no-op
No-operation instruction.

no-operation instruction
An instruction whose execution causes the computer to do nothing and then proceed to the next instruction to be executed.

NOR
A logic operator having the property that if P is a statement, Q is a statement, R is a statement, ..., then the NOR of P, Q, R, ... is TRUE if all statements are FALSE, FALSE if a least one statement is TRUE.

NOT
A logic operator having the property that if P is a statement, then NOT of P is TRUE if P is FALSE, FALSE if P is TRUE.

notation
A set of symbols, and the rules for their use in representation of data.

null string
A string containing no entity.

number
A mathematical entity that indicates quantity or amount of units.

numeral
A discrete representation of a number.

numeric
Pertaining to data or to physical quantities represented by numerals.

numeric data
Data represented by numerals.

object code
Output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

object program
A fully compiled or assembled program that is ready to be loaded into the computer.

octet
A byte composed of eight binary elements.

operand
An entity to which an operation is applied. That which is operated upon. An operand is usually identified by an address part of an instruction.

operating system
Software that controls the execution of computer programs and provides scheduling, debugging, input-output control, accounting, compilation, storage assignment, data management, and other related services.

operation

1. A well-defined action that, when applied to any permissible combination of known entities, produces a new entity.
2. A defined action, namely, the act of obtaining a result from one or more operands in accordance with a rule that completely specifies the result for any permissible combination of operands.
3. A program step undertaken or executed by a computer.
4. The event or specific action performed by a logic element.

operation code
A code used to represent the operations of a computer.

operator

1. A symbol that represents the action to be performed in a mathematical operation. In the description of a process, that which indicates the action to be performed on operands.
2. A person who operates a machine.

operator console
A functional unit containing devices that are used for communication between a computer operator and an automatic data processing system.

OR
A logic operator having the property that if P is a statement, Q is a statement, R is statement, ... then the OR of P, Q, R, ... is TRUE if at least one statement is TRUE, FALSE if all statements are FALSE.

output
Pertaining to a device, process, or channel involved in an output process, or to the data or states involved in an output process.

output area
An area of storage reserved for output.

output data
Data being delivered or to be delivered from a device or from a computer program.

overlay

1. In a computer program, a segment that is not permanently maintained in internal storage.
2. The technique of repeatedly using the same areas of internal storage during different stages of a program.
3. In the execution of a computer program, to load a segment of the computer program in a storage area previously occupied by parts of the computer program that are not currently needed.

padding
A technique that incorporates fillers in data.

page
A block of instructions, or data, or both, that can be located in main storage or in auxiliary storage. Segmentation and loading of these blocks is automatically controlled by a computer.

parameter
A variable that is given a constant value for a specified application that denotes the application.

parity bit
A check bit appended to an array of binary digits to make the sum of all the binary digits, including the check bit, always odd or always even.

parity check
A check that tests whether the number of ones (or zeros) in an array of binary digits is odd or even.

pass
One cycle of processing a body of data.

patch
To make an improvised modification. To modify a routine in an expedient way.

pointer
An identifier that indicates the location of an item of data.

position
In a string, each location that may be occupied by a character or binary element and identified by a serial number.

process
A course of events that occur according to an intended purpose or effect. A systematic sequence of operations to produce a specified result.

processor
A computer program that performs functions such as compiling, assembling, and translating for a specific programming language.

product
The number or quantity that results from multiplication.

program

1. A schedule or plan that specifies actions that may or may not be taken.
2. To design, write, and test computer programs.

program execution time
The interval during which the instructions of an object program are executed.

program library
An organized collection of computer programs that are sufficiently documented to allow them to be used by persons other than their authors.

programmer
A person who designs, writes, and tests computer programs.

programming
The designing, writing, and testing of computer programs.

programming language
An artificial language established for expressing computer programs.

pushdown list
A list that is constructed and maintained so that the next item to be retrieved is the most recently stored item in the list, for example last-in-first-out (LIFO). Synonymous with stack.

pushdown storage
A storage device that handles data in such a way that the next item to be retrieved is the most recently stored item still in the storage device; for example, last-in-first-out (LIFO).

pushup list
A list that is constructed and maintained so that the next item to retrieved is the earliest stored item still in the list, for example, first-in-first-out (FIFO).

pushup storage
A storage device that handles data in such a way that the next item to be retrieved is the earliest stored item still in the storage device; for example, first-in-first-out (FIFO).

qualified name
A data name explicitly accompanied by a specification of the class to which it belongs in a specified classification system.

queued access method
Any access method that synchronizes the transfer of data between the computer program using the access method and input-output devices, thereby minimizing delays for input-output operations.

quotient
The number or quantity that results from dividing the dividend by the divisor.

railroad syntax
A technique used to show how syntactically valid statements can be constructed.

random access
An access mode in which specific logical records are obtained from or placed into a mass storage file in a nonsequential manner.

range

1. The set of values that a quantity or function may take.
2. The difference between the highest and lowest value that a quantity or function may assume.

read
To acquire or to interpret data from a storage device, from a data medium, or from another source.

reading
The acquisition or interpretation of data from a storage device, from a data medium, or from another source.

real address
The address of an actual storage location in real storage.

real time

1. Pertaining to the actual time during which a physical process occurs.
2. Pertaining to the performance of a computation during the actual time that the related physical process occurs, in order that results of the computation can be used in guiding the physical process.

real-time processing

1. A mode of operation of a data processing system when performing real-time jobs.
2. The manipulation of data that are required or generated by some process while the process is in operation; usually the results are used to influence the process, and perhaps related processes, while it is occurring.

record
A collection of related data or words as a unit; for example, in stock control, each invoice could constitute one record.

record layout
The arrangement and structure of data or words in a record including the order and size of the components of the record.

record length
The number of characters forming a record.

recursive routine
A routine that may be used as a routine of itself, calling itself directly or being called by another routine, one that it itself has called. The use of a recursive routine or computer program usually requires the keeping of records of the status of its unfinished uses in, for example, a pushdown list.

recursive subroutine
A recursive subroutine that may be used as a subroutine of itself calling itself directly or being called by another subroutine, but one that it has called. The use of a recursive subroutine or computer program usually requires the keeping of records of the status of its unfinished uses in, for example, a pushdown list.

re-entrant code
A segment of object code that may be entered repeatedly and may be entered before any prior executions of the same segment of object code have been completed, and subject to the requirement that neither its external program parameters nor any instructions are modified during execution. A re-entrant segment of object code may be used simultaneously by more than one computer program simultaneously.

re-entrant program
A computer program that may be entered repeatedly and may be entered before any prior executions of the same computer program have been completed, and subject to the requirement that neither its external program parameters nor any instructions are modified during execution. A re-entrant program may be used simultaneously by more than one computer program.

re-entrant routine
A routine that may be entered repeatedly and may be entered before any prior executions of the same routine have been completed, and subject to the requirement that neither its external program parameters nor any instructions are modified during execution. A re-entrant routine may be used simultaneously by more than one computer program.

re-entrant subroutine
A subroutine that may be entered repeatedly and may be entered before any prior executions of the same subroutine have been completed, and subject to the requirement that neither its external program parameters nor any instructions are modified during execution. A re-entrant subroutine may be used by more than one computer program simultaneously.

re-entry point
: The address or the label of the instruction at which the computer program that called a subroutine is re-entered from the subroutine.

register
: In a computer, a storage device usually intended for some special purpose, capable of storing a specified amount of data such as a bit or a word.

relative address
: An address expressed as a difference with respect to a base address.

relocatable address
: An address that is adjusted when the computer program containing it is relocated.

relocate
: To move a computer program or part of a computer program, and to adjust the necessary address references so that the computer program can be executed after being moved.

reserved word
: A word of a source language having meaning fixed by rules of that language and which cannot be altered for the convenience of any one computer program expressed in the source language. Computer programs expressed in the source language may be prohibited from using reserved words in other contexts.

reset
: To cause a counter to take the state that corresponds to a specified initial number.

restart
: The resumed execution of a computer program that uses data recorded at a checkpoint.

result
: An entity produced by the performance of an operation.

return
: With a subroutine, to bind a variable in the computer program that called the subroutine or to effect a link to the computer program that called the subroutine.

right-justify

1. To shift the contents of a register so that the character at the right-hand end of the data within the register is moved to a specified position in the register.
2. To control the positions of characters on a page so that the right-hand margin of printing is regular.
3. To align characters horizontally so that the rightmost character of a string is in a specified position.

roll-in
: To restore in main storage, data or one or more computer programs that were previously rolled out.

roll-out
: To transfer data or one or more computer programs from main storage to auxiliary storage for the purpose of freeing main storage for another use.

round
To delete or omit one or more of the least significant digits in a positional representation and to adjust the part retained according to a specified rule. The purpose of rounding is usually to limit the precision of the numeral or to reduce the number of characters in the numeral, or to do both.

routine
An ordered set of instructions that may have some general or frequent use.

run

1. A single performance of one or more jobs.
2. A single, continuous performance of a computer program or routine.

running time
The elasped time taken for the execution of a computer program.

scalar
A quantity characterized by a single number.

scope
The scope of a procedure is determined at compile time by the SDL/UPL compiler and is the range within a program over which an identifier or procedure identifier can be referenced.

The scope of an identifier is a direct result of the lexic level of procedures and the storage allocation techniques used by the SDL/UPL compiler. The scope of an identifier is that portion of the SDL/UPL program which can reference the identifier.

SDL
The acronym for Software Development Language.

search

1. The examination of a set of items for one or more items having a given property.
2. To examine a set of items for one or more having a given property.

search key
In the conduct of a search, the data to be compared to a specified part of each item.

sector
A part of a track or band on a magnetic drum, magnetic disk, or disk pack.

seek
To selectively position the access mechanism of a direct access device.

segment
A self-contained portion of a computer program that may be executed without the entire computer program necessarily being maintained in internal storage at any one time.

self-relative address
A relative address that uses the address of the instruction in which it appears as the base address.

self-relative addressing
A method of addressing in which the address part of an instruction contains a self-relative address.

semantics

1. The relationships of characters or groups of characters to their meanings, independent of the manner of their interpretation and use.
2. The relationships between symbols and their meanings.

sequence

1. A series of items that have been sequenced.
2. An arrangement of items according to a specified set of rules. For example, items arranged alphabetically, numerically, or chronologically.

serial access

1. The facility to obtain data from a storage device or to enter data into a storage device in such a way that the process depends on the location of that data and on a reference to data previously accessed.
2. Pertaining to the sequential or consecutive transmission of data to or from storage.

set

1. A finite or infinite number of objects of any kind, of entities, or of concepts, that have a given property or properties in common.
2. To cause a counter to take the state corresponding to a specified number.
3. To place a storage device into a specified state, usually other than that denoting zero.

sign bit
A bit or a binary element that occupies a sign position and indicates the algebraic sign of the number represented by the numeral with which it is associated.

sign character
A character that occupies a sign position and indicates the algebraic sign of the number represented by the numeral with which it is associated.

sign digit
A digit that occupies a sign position and indicates the algebraic sign of the number represented by the numeral with which it is associated.

significant digit
In a numeral, a digit that is needed for a given purpose; in particular, a digit that must be kept to preserve a given accuracy or a given precision.

sign position
A position, normally located at one end of a numeral, that contains an indicator denoting the algebraic sign of the number represented by the numeral.

skip

1. To ignore one or more instructions in a sequence of instructions.
2. To pass over one or more positions on a data medium, for example, to perform one or more line feed operations.

software
Computer programs, procedures, rules, and other documentation concerned with the operation of a data processing system.

Software Development Language (SDL)
A B 1000 programming language that is used to write B 1000 system software.

source language
A language from which statements are translated.

source program
A computer program expressed in a source language.

space

1. A site intended for the storage of data; for example, a site on a printed page or a location in a storage medium.
2. A basic unit of area, usually the size of a single character.
3. One or more space characters.
4. To advance the reading or display position according to a prescribed format: for example, to advance the printing or display position horizontally to the right or vertically down.

span
The difference between the highest and the lowest values that a quantity or function may take.

special character
A graphic character in a character set that is not a letter, digit, or a space character.

stack
Synonym for pushdown list.

statements
Meaningful expressions that describe or specify operations which are complete in the context of the programming language.

step

1. One operation in a computer routine.
2. To cause a computer to execute one operation.

stop instruction
An exit that specifies the termination of the execution of a computer program.

storage

1. The action of placing data into a storage device and retaining it for subsequent use.
2. The retention of data in a storage device.

store

1. To enter data into a storage device or to retain data in a storage device.
2. In computer programming, to copy data from registers into internal storage.

string
A linear sequence of entities such as characters or physical elements.

structured programming
The art of combining logically independent algorithms to solve complex problems.

subroutine

1. A sequenced set of statements that may be used in one or more computer programs and at one or more points in a computer program.
2. Part of another routine.

subroutine call
The subroutine, in object coding, that performs the call function.

subscript
A symbol associated with the name of a set to identify a particular subset or element.

subset
A set, each element of which is an element of a specified other set.

subtrahend
In a subtraction operation, the number or quantity subtracted from the minuend.

sum
The number or quantity that is the result of the addition of two or more numbers or quantities.

supervisory program
A computer program, usually part of an operating system, that controls the execution of other computer programs and regulates the flow of work in a data processing system.

supervisory routine
A routine, usually part of an operating system, that controls the execution of other routines and regulates the flow of work in a data processing system.

switch

1. In a computer program, a parameter that controls branching and is bound prior to the branch-point being reached.
2. A device or programming technique for making a selection; for example, a toggle, a conditional jump.

switch indicator
In computer programming, an indicator that determines or shows the setting of a switch.

symbol

1. A conventional representation of a concept or a representation of a concept upon which agreement has been reached.
2. A representation of something by reason of relationship, association, or convention.

syntax

1. The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use.
2. The structure of expressions in a language.
3. The rules governing the structure of a language.
4. The relationships among symbols.

system
In data processing, a collection of people, machines, and methods organized to accomplish a set of specific functions.

table

1. An array of data, each item of which is unambiguously identified by means of one or more arguments.
2. A collection of data in which each item is uniquely identified by a label, by its position relative to the other items, or by some other means.

table lookup
A procedure for obtaining the value corresponding to an argument from a table of values.

tag
One or more characters, attached to a set of data that contains information about the set, including its identification.

task

1. The basic unit of work from the standpoint of a control program.
2. In a multiprogramming or multiprocessing environment, a computer program, or portion thereof, capable of being specified to the control program as a unit of work. Tasks compete for system resources.

trace
A record of the execution of a computer program; it exhibits the sequences in which the instructions were executed.

trailing zero
In positional notation, a zero in a less significant digit place than the digit place of the least significant nonzero digit of a numeral.

transfer
To send data from one place and to receive the data at another place.

translate
To transform data from one language to another.

transmission

1. The sending of data to one or more locations or recipients.
2. The sending of data from one place for reception elsewhere.
3. In ASCII and data communication, a series of characters including headings and texts.

transmit
To send data from one place for reception elsewhere.

truncate
To terminate a computational process in accordance with some rule. For example, to end the evaluation of a power series at a specified term.

truncation

1. The deletion or omission of a leading or of a trailing portion of a string in accordance with specified criteria.
2. The termination of a computation process, before any final conclusion or natural termination, according to specified rules.

unary operation
An operation with one and only one operand.

unary operator
An operator that represents an operation on one and only one operand.

unit

1. A device having a special function.
2. A basic element.

UPL
The acronym for User Programming Language.

User Programming Language (UPL)
A B 1000 computer system language that is a subset of the B 1000 Software Development Language (SDL).

variable

1. A character or group of characters which refer to a value and which, in the execution of a computer program, correspond to an address.
2. A quantity that can assume any of the given set of values.

variable-length record
Pertaining to a file in which the records need not be uniform in length.

virtual address
The address of a storage location in virtual storage.

virtual memory
See virtual storage.

virtual storage
Space on storage devices which is used as main storage (by the user of) a computing system, and in which virtual addresses are mapped into real addresses. The size of the storage is limited only by the addressing scheme of the computing system and by the amount of auxiliary storage available, rather than by the actual number of main storage locations.

write
To make a permanent or temporary recording of data in a storage device or on a data medium.

writing
The action of making a permanent or temporary recording of data in a storage device or on a data medium.

zero
In data processing, the number which does not alter the value of another number through addition or subtraction.

zerofill
To character fill with representation of the character zero.

zero suppression
The elimination of zeros from a numeral to which they have no significance. Zeros that have no significance include t the left of the nonzero digits in the integral part of a numeral and those to the right of the nonzero digits in the fractional part.

INDEX

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)