IBM

VS FORTRAN Version 2

SC26-4221-3

Language and Library Reference

Release 3

$\sinh 3x = \sinh x(4 \cosh^2 x - 1)$
$\sinh 4x = \sinh x \cosh x(8 \cosh^2 x - 4)$
$\sinh 5x = \sinh x(1 - 12 \cosh^2 x + 16 \cosh^4 x)$
$\cosh 3x = \cosh x(4 \cosh^2 x - 3)$
$\cosh 4x = 1 - 8 \cosh^2 x + 8 \cosh^4 x$
$\cosh 5x = \cosh x(5 - 20 \cosh^2 x + 16 \cosh^4 x)$
a^1
a^2
c^{10}
c^{21}
$x = \frac{2y-b-a}{b-a}$
$f(1+a^2)^{1/2}$

IBM

VS FORTRAN Version 2

SC26-4221-3

Language and Library Reference

Release 3

Fourth Edition (March 1988)

This edition replaces and makes obsolete the previous edition, SC26-4221-2.

This edition applies to Release 3 of VS FORTRAN Version 2, Program numbers 5668-806 and 5668-805, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Changes" following the preface "About This Book." Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent publication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, 4300, and 9370 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's program may be used. Any functionally equivalent program may be used instead.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, P.O. Box 49023, Programming Publishing, San Jose, California, U.S.A. 95161-9023. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

About This Book

This book contains reference information about VS FORTRAN Version 2. It is not intended as a tutorial. Rather, it is designed as a reference tool for the user who already has some basic FORTRAN knowledge.

How This Book Is Organized

- **"Part 1. Language Reference,"** gives the programming rules for the VS FORTRAN Version 2 source language.
  - **Chapter 1, "Language,"** describes elements of the FORTRAN language: statements, comments, syntax, and other conventions used to convey information to the compiler.
  - **Chapter 2, "Data,"** discusses the constants, variables and arrays that you can use with VS FORTRAN Version 2.
  - **Chapter 3, "Expressions,"** explains the four kinds of expressions: arithmetic, character, relational, and logical.
  - **Chapter 4, "Statements,"** presents the syntax rules and conventions of the VS FORTRAN Version 2 language statements.
  - **Chapter 5, "Intrinsic Functions,"** contains usage information for the explicitly-called routines commonly used for mathematical computations and character and bit conversions.
- **"Part 2. Library Reference,"** provides information about the subroutines and functions that are supplied with the product.
  - **Chapter 6, "Mathematical, Character, and Bit Routines,"** contains usage information about the implicitly-called routines commonly used for computations and conversions.
  - **Chapter 7, "Service Subroutines,"** provides you with usage information concerning the subroutines that have been provided for general programming tasks.
  - **Chapter 8, "Data-in-Virtual Subroutines,"** presents information on the subroutines that allow you to use the Data-in-Virtual facility on MVS/XA.
  - **Chapter 9, "Extended Error-Handling Subroutines and Error Option Table,"** gives you information about the subroutines that are used for extended error handling.
  - **Chapter 10, "Multitasking Facility (MTF) Subroutines,"** presents information on the subroutines that allow you to use the Multitasking Facility when running under MVS/XA.
- **Appendixes**
  - **Appendix A, "Source Language Flaggers,"** lists the items which are flagged when either the FIPS compiler option or the SAA compiler option is specified.
  - **Appendix B, "Assembler Language Information,"** provides information on using the VS FORTRAN Version 2 mathematical and service routines in assembler language programs.

- **Appendix C, "Sample Storage Printouts,"** presents the output format of symbolic dumps, including output examples for variable items, for array items, and for non-recoverable failure.
- **Appendix D, "Library Procedures and Messages,"** contains the explanations of the program-interrupt and error procedures used by the VS FORTRAN Version 2 library.

How to Use This Book

For the task of application programming, you will need to use both this book and *VS FORTRAN Version 2 Programming Guide.* Whereas this book contains detailed information on the VS FORTRAN Version 2 language and library, the *VS FORTRAN Version 2 Programming Guide* contains information on how to compile and run your VS FORTRAN Version 2 programs, as well as some information on advanced coding topics.

Syntax Notation

The following items explain how to interpret the syntax used in this manual:

- Uppercase letters and special characters (such as commas and parentheses) are to be coded exactly as shown, except where otherwise noted. You can, however, mix lowercase and uppercase letters; lowercase letters are equivalent to their uppercase counterparts, except in character constants.
- Italicized, lowercase letters or words indicate variables, such as array names or data types, and are to be substituted.
- Underlined letters or words indicate IBM-supplied defaults.
- Ellipses (...) indicate that the preceding optional items may appear one or more times in succession.
- Braces ({ }) group items from which you must choose one.
- Square brackets ([ ]) group optional items from which you may choose none, one, or more.
- OR signs (|) indicate you may choose only one of the items they separate.
- Blanks in FORTRAN statements are used to improve readability; they have no significance, except when shown within apostrophes (' '). In non-FORTRAN statements, blanks may be significant. Code non-FORTRAN statements exactly as shown.
- For clarity of presentation, continuation designators have been omitted from continuation lines in examples.

For example, given the following syntax:

Syntax

CALL *name* **[([***arg1* **[,***arg2***] ...])]**

these statements are among those allowed:

```
CALL ABCD
CALL ABCD ()
CALL ABCD (X)
CALL ABCD (X, Y)
CALL ABCD (X, Y, Z)
```

For double-byte character data, the following syntax notation is used:

<	represents the shift-out character (X'0E'), which indicates the start of double-byte character data
>	represents the shift-in character (X'0F'), which indicates the end of double-byte character data
.	represents the first byte (X'42') of an EBCDIC double-byte character
kk	represents a double-byte character not in the EBCDIC double-byte character set

In examples, the character b represents a blank.

Summary of the VS FORTRAN Version 2 Publications

The following table lists the VS FORTRAN Version 2 publications and the tasks they support.

Task	VS FORTRAN Version 2 Publications	Order Numbers
Evaluation and Planning	General Information Licensed Program Specifications	GC46-4219 GC26-4225
Installation and Customization	Installation and Customization for VM Installation and Customization for MVS	SC26-4339 SC26-4340
Application Programming	Language and Library Reference Programming Guide Interactive Debug Guide and Reference Reference Summary	SC26-4221 SC26-4222 SC26-4223 SX26-3751
Diagnosis	Diagnosis Guide	LY27-9516

Industry Standards

The VS FORTRAN Version 2 Compiler and Library licensed program is designed according to the specifications of the following industry standards, as understood and interpreted by IBM as of March, 1988.

The following two standards are technically equivalent. In the publications, references to **FORTRAN 77** are references to these two standards:

- American National Standard Programming Language FORTRAN, ANSI X3.9-1978 (also known as FORTRAN 77)
- International Organization for Standardization ISO 1539-1980 Programming Languages-FORTRAN

The bit string manipulation functions are based on ANSI/ISA-S61.1.

The following two standards are technically equivalent. References to **FORTRAN 66** are references to these two standards:

- American Standard FORTRAN, X3.9-1966
- International Organization for Standardization ISO R 1539-1972 Programming Languages-FORTRAN

At both the FORTRAN 77 and the FORTRAN 66 levels, the VS FORTRAN Version 2 language also includes IBM extensions. References to **current FORTRAN** are references to the FORTRAN 77 standard, plus the IBM extensions valid with it. References to **old FORTRAN** are references to the FORTRAN 66 standard, plus the IBM extensions valid with it.

Documentation of IBM Extensions

In addition to the statements available in FORTRAN 77, IBM provides "extensions" to the language. These extensions are printed in color, as in the following sentence:

This sentence shows how IBM language extensions in text are documented.

Summary of Changes

Release 3, March 1988

Major Changes to the Product

- Enhancements to the vector feature of VS FORTRAN Version 2
  - Automatic vectorization of user programs is improved by relaxing some restrictions on vectorizable source code. Specifically, VS FORTRAN Version 2 can now vectorize MAX and MIN intrinsic functions, COMPLEX compares, more adjustably dimensioned arrays, and more DO loops with unknown increments.
  - Ability to specify certain vector directives globally within a source program.
  - Addition of an option to generate the vector report in source order.
  - Ability to collect tuning information for vector source programs.
    - Ability to record compile-time statistics on vector length and stride and include these statistics in the vector report.
    - Ability to record and display run-time statistics on vector length and stride. Two new commands, VECSTAT and LISTVEC, have been added to Interactive Debug to support this function.
    - Enhancements to Interactive Debug to allow timing and sampling of DO loops.
    - Inclusion of vector feature messages in the on-line HELP function of Interactive Debug.
  - Simplification of the VECTOR compile-time option.
  - Vectorization is allowed at OPTIMIZE(2) and OPTIMIZE(3).
  - Changes to the way in which the vector feature treats partial sum processing result in a performance improvement.
- Enhancements to the language capabilities of VS FORTRAN Version 2
  - Ability to specify the file or data-set name on the INCLUDE statement.
  - Ability to write comments on the same line as the code to which they refer.
  - Support for the DO WHILE programming construct.
  - Support for the END DO statement as the terminal statement of a DO loop.
  - Enhancements to the DO statement so that the label of the terminal statement is optional.
  - Support for statements extending to 99 continuation lines or a maximum of 6600 characters.

  - Implementation of IBM's Systems Application Architecture (SAA) FORTRAN definition; support for a flagger to indicate source language that does not conform to the language defined by SAA.
  - Support for the use of double-byte characters as variable names and as character data in source programs, I/O, and for Interactive Debug input and output.
  - Support for the use of a comma to indicate the end of data in a formatted input field, thus eliminating the need for the user to insert leading or trailing zeros or blanks.
- Enhancements to the programming aids in VS FORTRAN Version 2
  - Enhancements to the intercompilation analysis function to detect conflicting and undefined arguments.
  - Support for the Data-In-Virtual facility of MVS/XA.
  - Ability to allocate certain commonly used files and data sets dynamically.
  - Enhancements to the Multitasking Facility to allow large amounts of data to be passed between parallel subroutines using a dynamic common block.
  - Support for named file I/O in parallel subroutines using the Multitasking Facility.
  - Ability to determine the amount of CPU time used by a program or a portion of a program by using the CPUTIME service subroutine.
  - Ability to determine the FORTRAN unit numbers that are available by using the UNTANY and UNTNOFD service subroutines.
- Enhancements to the full screen functions of Interactive Debug

Major Changes to This Manual

- Documentation of the major product enhancements has been added.
- Reorganization of Chapter 7, "Service Subroutines" on page 269
- Addition of Chapter 8, "Data-in-Virtual Subroutines" on page 293
- Reorganization of Appendix A, "Source Language Flaggers" on page 339
- Removal of the appendix, "IBM and ANS FORTRAN Features"
- Consolidation of the appendix, "EBCDIC and ISCII/ASCII Codes" to Figure 2 on page 6
- Removal of the appendix, "Algorithms for Mathematical Functions"
- Removal of the appendix, "Accuracy Statistics"

Release 2, June 1987

Major Changes to the Product

- Support for 31-character symbolic names, which can include the underscore (_) character.
- The ability to detect incompatibilities between separately-compiled program units using the new compile-time option ICA (intercompilation analyzer).
- Addition of the NONE keyword for the IMPLICIT statement.
- Enhancement of SDUMP when specified for programs vectorized at LEVEL(2), so that ISNs of vectorized statements and DO-loops appear in the object listing.
- The ability of run-time library error-handling routines to identify vectorized statements when a program interrupt occurs, and the ability under Interactive Debug to set breakpoints at vectorized statements.
- Under MVS, addition of a data set and an optional DD statement to be used during execution for loading library modules and Interactive Debug.
- Under VM, the option of creating a combined VSF2LINK TXTLIB during installation for use in link mode in place of VSF2LINK and VSF2FORT.
- The ability to sample CPU use within a program unit using the new Interactive Debug commands LISTSAMP and ANNOTATE.
- The ability to reset a closed unit to its original (preconnected) state using the new Interactive Debug command RECONNECT.
- The ability to automatically allocate data sets for viewing in the Interactive Debug source window.
- Change in the relative placement in storage of local items, that is, those that are not in common blocks. Programs which depend upon a given arrangement of items in storage may have to be re-coded. This change does not affect items in common blocks nor the association of equivalenced items.
- Change in semantics for OPEN, CLOSE, and INQUIRE statements and the addition of the execution-time options OCSTATUS and NOOCSTATUS.
  - The INQUIRE statement can be used to determine the properties of a file that has never been opened. INQUIRE specifiers SEQUENTIAL, DIRECT, KEYED, FORMATTED, and UNFORMATTED will return values dependent on how the files could potentially be connected.
  - If execution-time option OCSTATUS is in effect:
    - File existence is checked for consistency with the OPEN statement specifiers STATUS='OLD' and STATUS='NEW'.
    - File deletion occurs when the CLOSE statement specifier STATUS='DELETE' is given (on devices which allow deletion).
    - A preconnected file is disconnected when a CLOSE statement is given or when another file is opened on the same unit. It can be reconnected only by an OPEN statement when there is no other file currently connected to that unit.

- If execution-time option NOOCSTATUS is in effect:
  - File existence is not checked for consistency with the OPEN statement specifiers STATUS='OLD' and STATUS='NEW'.
  - File deletion does not occur with the CLOSE statement specifier STATUS='DELETE'.
  - A preconnected file is disconnected when a CLOSE statement is given or when another file is opened on the same unit. It can be reconnected by a sequential READ or WRITE, BACKSPACE, OPEN, REWIND, or ENDFILE statement when there is no other file currently connected to that unit.
- An OPEN statement cannot be issued for a currently open file except to change the BLANK specifier.

Major Changes to This Manual

Documentation of the above product enhancements has been added.

Release 1.1, September 1986

Major Changes to the Product

- Addition of vector directives, including compile-time option (DIRECTIVE) and installation-time option (IGNORE)
- Addition of NOIOINIT execution-time option
- Addition of support for VM/XA System Facility Release 2.0 (5664-169) operating system

Major Changes to This Manual

None. However, clarifications and editorial changes have been made throughout.

Contents

Part 1. Language Reference 1

Chapter 1. Language 3
Valid and Invalid Programs 3
Language Definitions 3
Language Syntax 4
  Input Records 4
  Source Statement Characters 5
  Names 7
  Source Language Statements 8
  Comments 12
  Statement Labels 13
  Keywords 13

Chapter 2. Data 15
Constants 15
  Arithmetic Constants 15
  Logical Constants 19
  Character Constants 19
  Hollerith Constants 20
  Hexadecimal Constants 21
Variables 22
  Variable Names 22
  Data Types and Lengths 22
Array 24
  Subscripts 25
  Size and Type Declaration of an Array 26
Character Substrings 28

Chapter 3. Expressions 31
Evaluation of Expressions 31
Arithmetic Expressions 32
  Arithmetic Operators 32
  Rules for Constructing Arithmetic Expressions 32
  Use of Parentheses in Arithmetic Expressions 34
  Type and Length of the Result of Arithmetic Expressions 34
  Examples of Arithmetic Expressions 36
Character Expressions 37
  Use of Parentheses in Character Expressions 38
Relational Expressions 38
Logical Expressions 40
  Logical Operators 40
  Order of Computations in Logical Expressions 42
  Use of Parentheses in Logical Expressions 43

Chapter 4. Statements 45
Statement Categories 45
  Assignment Statements 45
  Control Statements 45

DATA Statement 46
Debug Statements 46
Input/Output Statements 46
PROGRAM Statement 48
Specification Statements 48
Subprogram Statements 48
Compiler Directives 48
Order of Statements in a Program Unit 49
Statement Descriptions 50
Arithmetic IF Statement 50
ASSIGN Statement 51
Assigned GO TO Statement 52
Assignment Statements 52
AT Statement 56
BACKSPACE Statement 57
BLOCK DATA Statement 59
Block IF Statement 60
CALL Statement 60
CHARACTER Type Statement 63
CLOSE Statement 64
Comments 66
COMMON Statement 66
COMPLEX Type Statement 68
Computed GO TO Statement 68
CONTINUE Statement 68
DATA Statement 69
DEBUG Statement 71
DELETE Statement 75
DIMENSION Statement 76
DISPLAY Statement 77
DO Statement 77
DOUBLE PRECISION Type Statement 82
DO WHILE Statement 82
EJECT Statement 83
ELSE Statement 83
ELSE IF Statement 83
END Statement 83
END DEBUG Statement 84
END DO Statement 84
ENDFILE Statement 85
END IF Statement 86
ENTRY Statement 86
EQUIVALENCE Statement 89
Explicit Type Statement 91
EXTERNAL Statement 95
FORMAT Statement 96
FUNCTION Statement 120
GO TO Statements 125
IF Statements 127
IMPLICIT Statement 133
INCLUDE Statement 134
INQUIRE Statement 136
INTEGER Type Statement 147
INTRINSIC Statement 147
Logical IF Statement 148

- LOGICAL Type Statement 148
- NAMELIST Statement 148
- OPEN Statement 151
- PARAMETER Statement 158
- PAUSE Statement 159
- PRINT Statements 160
- PROGRAM Statement 162
- READ Statements 163
- REAL Type Statement 195
- RETURN Statement 196
- REWIND Statement 198
- REWRITE Statement—Formatted with Keyed Access 199
- REWRITE Statement—Unformatted with Keyed Access 201
- SAVE Statement 203
- Statement Function Statement 204
- Statement Labels 207
- STOP Statement 207
- SUBROUTINE Statement 208
- TRACE OFF Statement 213
- TRACE ON Statement 213
- Unconditional GO TO 213
- WAIT Statement 214
- WRITE Statements 216

Chapter 5. Intrinsic Functions 243

Part 2. Library Reference 261

Chapter 6. Mathematical, Character, and Bit Routines 263

- Explicitly Called Routines 263
- Implicitly Called Routines 263
- Alternative Mathematical Library Subroutines 263

Chapter 7. Service Subroutines 269

- Mathematical Exception Test Subroutines 270
  - DVCHK Subroutine 270
  - OVERFL Subroutine 270
- Storage Dump Subroutines 271
  - DUMP/PDUMP Subroutines 271
  - CDUMP/CPDUMP Subroutines 273
  - SDUMP Subroutine 274
- Return Code Subroutines 276
  - SYSRCS Subroutine 276
  - SYSRCT Subroutine 276
  - SYSRCX Subroutine 276
- Other Service Subroutines 277
  - ASSIGNM Subroutine 277
  - CLOCK/CLOCKX Subroutines 278
  - CPUTIME Subroutine 280
  - DATIM/DATIMX Subroutines 282
  - EXIT Subroutine 283
  - FILEINF Subroutine 283
  - SYSABN/SYSABD Subroutines 287
  - UNTNOFD/UNTANY Subroutines 288

XUFLOW Subroutine 291

Chapter 8. Data-in-Virtual Subroutines 293
Overview 293
Fixed-View Versus Varying-View Subroutines 294
Syntax of the Subroutines 296
Syntax of Fixed-View Subroutines 296
Syntax of Varying-View Subroutines 300
Syntax of Common Subroutines 305
Interface to the Data-in-Virtual Functions 306
Sample Program with Fixed-View Subroutines 307
Sample Program with Varying-View Subroutines 308
Remapping a Dynamic Common to Different Parts of the Data Object . . . 312
Resetting a Dynamic Common 313
Ensuring Data Integrity 313
Performance and Storage Factors 313
Effect on Optimization 314
Using Data-in-Virtual in an MTF Environment 314

Chapter 9. Extended Error-Handling Subroutines and Error Option Table 315
Extended Error Handling 315
Error-Handling Subroutines 316
ERRMON Subroutine 316
ERRSAV Subroutine 317
ERRSET Subroutine 318
ERRSTR Subroutine 321
ERRTRA Subroutine 321
Error Option Table 322

Chapter 10. Multitasking Facility (MTF) Subroutines 335
NTASKS Subroutine 335
DSPTCH Subroutine 336
SYNCRO Subroutine 337
SHRCOM Subroutine 338

Appendix A. Source Language Flaggers 339
ANS Language (FIPS) Flagger 339
Systems Application Architecture Flagger 339
Items Flagged for FIPS and SAA 339

Appendix B. Assembler Language Information 343
Library Availability 343
Initializing the Execution Environment 344
Calling Sequences 344
Assembler Language Calling Sequence 346
Supplying Correct Parameters 346
Mathematical Routine Results 347
Space Considerations 347
Floating Point Registers Used by Routines 348
Calling Vector Intrinsic Elementary Functions 351
Required Contents of Registers 352
Program Mask 353
Error Handling 353
Vector Registers Used by Vector Intrinsic Elementary Functions 353

Appendix C. Sample Storage Printouts 357
Output from Symbolic Dumps 357
Output Format 357
Variable Noncharacter 357
Variable Character 358
Array 358
Control Flow Information 359
I/O Unit Information 360
I/O Unit Status Information 361
Examples of Sample Programs and Symbolic Dump Output 361

Appendix D. Library Procedures and Messages 375
Library Interruption Procedures 375
Library Error Procedures 375
Library Messages 375
Operator Messages 376
Program-Interrupt Messages 377
Execution Error Messages 379

Glossary 439

Index 447

Part 1. Language Reference

The following topics are discussed in Part 1:

Language

Data

Expressions

Statements

Intrinsic Functions

Chapter 1. Language

A valid FORTRAN program is made up of three basic elements:

Data Consists of constants, variables, and arrays. See Chapter 2, "Data" on page 15.

Expressions Executable sets of arithmetic, character, logical, or relational data. See Chapter 3, "Expressions" on page 31.

Statements Combinations of data and expressions. See Chapter 4, "Statements" on page 45.

Valid and Invalid Programs

This manual defines the rules (that is, the syntax, semantics, and restrictions) applicable for writing valid FORTRAN programs, either for FORTRAN 77 or for FORTRAN 77 plus IBM extensions. Most violations of language rules are diagnosed by the compiler. However, some syntactic and semantic combinations are not diagnosed until run time. Programs that contain these undiagnosed combinations are invalid programs, whether or not they run as expected.

Language Definitions

Some of the terms used in the discussion of the FORTRAN programming language are defined as follows:

Program unit. A sequence of statements and optional comment lines, with the final statement being an END statement, constituting a main program or subprogram.

Main program. The program unit that receives control from the system when the executable program is invoked at run time. A main program may contain any statement except BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY, or RETURN.

Function Subprogram. A program unit that specifies a function. It is invoked by a function reference and returns a value to the invoking program unit. The first statement of a function subprogram must be a FUNCTION statement. It may contain any statement except PROGRAM, SUBROUTINE, and BLOCK DATA.

Subroutine Subprogram. A program unit that is invoked by its name or one of its entry names in a CALL statement. The first statement of a subroutine subprogram must be a SUBROUTINE statement. It may contain any statement except PROGRAM, FUNCTION, and BLOCK DATA.

Block Data Subprogram. A program unit that provides initial values for variables and array elements in named common blocks. The first statement of a block data subprogram must be a BLOCK DATA statement. The only other statements that may appear in a block data subprogram are DIMENSION, EQUIVALENCE, COMMON, type, IMPLICIT, PARAMETER, SAVE, DATA, and END. Comment lines are permitted.

Procedure. A sequenced set of statements that may be invoked by a program unit to perform a particular activity.

Intrinsic function. A function, supplied by VS FORTRAN Version 2, that performs mathematical, character, logical, or bit-manipulation operations. (See "INTRINSIC Statement" on page 147 and Chapter 5, "Intrinsic Functions" on page 243).

External procedure. A subroutine or function subprogram written in FORTRAN or in a language accessible by VS FORTRAN Version 2.

Executable program. A collection of program units consisting of one main program and, optionally, one or more subprograms.

Executable statement. A statement that moves data, performs an arithmetic, character, logical, or relational operation, or alters the sequential processing of statements.

Nonexecutable statement. A statement that describes the characteristics of the program unit, of data, of editing information, or of statement functions, but does not cause an action to be taken by the program.

Preconnected file or unit. A file or unit is preconnected if, at the beginning of program processing, it is connected. For further information on connection and preconnection, see "Input/Output Semantics" on page 46. The terms file connection and unit connection are equivalent.

Additional definitions can be found in the "Glossary" on page 439.

Language Syntax

The meaning of a program unit is determined from keywords, special characters, and rules that group these keywords and characters together to form source language statements. For the compiler to process its input, certain syntax rules must be carefully adhered to when entering the following items:

Source statement characters
Names
Source language statements
Statement labels
Keywords

Input Records

VS FORTRAN Version 2 accepts source input in either of two formats:

- Free-form input format
- Fixed-form input format

A program unit must be written in either free form or fixed form, not both. For more information, see the FREE | FIXED compiler option in *VS FORTRAN Version 2 Programming Guide.* For a detailed description of the use and implementation of the two formats, see "Free-Form Input Format" on page 9 and "Fixed-Form Input Format" on page 11.

The compiler receives its input in fixed-length (80 byte), variable-length, or undefined-length records. For variable or undefined-length records, the compiler accepts a maximum record length of 80 bytes.

Source Statement Characters

The VS FORTRAN character set, shown in Figure 1, is made up of letters, digits and special characters.

Letters				Digits	Special Characters	
A	N	a	n	0		Blank
B	O	b	o	1	.	Decimal point (period)
C	P	c	p	2	(	Left parenthesis
D	Q	d	q	3	+	Plus sign
E	R	e	r	4	!	Exclamation point
F	S	f	s	5	*	Asterisk
G	T	g	t	6	)	Right parenthesis
H	U	h	u	7	-	Minus sign
I	V	i	v	8	/	Slash
J	W	j	w	9	,	Comma
K	X	k	x		_	Underscore
L	Y	l	y		:	Colon
M	Z	m	z		'	Apostrophe
					=	Equals sign
$ Currency symbol					"	Quotation mark

Figure 1. Source Statement Characters (Character Set)

The following table, Figure 2 on page 6, shows the two-digit hexadecimal code points for the characters in the character set. To determine the hexadecimal code point of a character, look at the top of the column for the first hex digit, and at the left of the row for the second hex digit.

In addition to the characters shown, the VS FORTRAN character set also includes the shift-out character (X'0E') and the shift-in character (X'0F'). The empty spaces in the table are subject to implementation; they are not part of the VS FORTRAN character set.

In statements, lowercase letters are equivalent to their uppercase counterparts, except within the following:

- Character constants
- H and apostrophe edit descriptors

In statements, blanks and double-byte blanks are significant only in the following:

- Character constants
- H and apostrophe edit descriptors
- The count of characters permitted in a statement

You may use blanks anywhere else within a program unit to make it more readable.

Hex Digit 1st→ 2nd↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	ƀ		-									0
-1			/		a	j			A	J		1
-2					b	k	s		B	K	S	2
-3					c	l	t		C	L	T	3
-4					d	m	u		D	M	U	4
-5					e	n	v		E	N	V	5
-6					f	o	w		F	O	W	6
-7					g	p	x		G	P	X	7
-8					h	q	y		H	Q	Y	8
-9					i	r	z		I	R	Z	9
-A		!		:								
-B	.	$	,									
-C		*		@								
-D	(	)	_	'								
-E	+			=								
-F				"								

Figure 2. EBCDIC Source Statement Code Points

Note: Hex'40' is the blank character.

Double-byte characters may be used in symbolic names and in character constants when the DBCS compiler option is in effect. For information on the DBCS compiler option, see *VS FORTRAN Version 2 Programming Guide*. They may be used in comment lines at all times.

Double-byte characters are represented by a two-byte code, where each byte is in the range of X'41' to X'FE', except for the double-byte blank, which has an internal representation of X'4040'.

An EBCDIC double-byte character is one with X'42' as the first byte and the hexadecimal equivalent of a letter, digit, or special character as the second byte. For example, an EBCDIC double-byte capital letter A is represented internally as X'42C1' (because the hexadecimal equivalent of A is X'C1').

Names

Names can be used to identify the following items in a program unit:

- An array (see "Array" on page 24)
- A variable (see "Variables" on page 22)
- A constant (see "PARAMETER Statement" on page 158)
- A main program (see "PROGRAM Statement" on page 162)
- A statement function (see "Statement Function Statement" on page 204)
- An intrinsic function (see "INTRINSIC Statement" on page 147)
- A function subprogram (see "FUNCTION Statement" on page 120)
- A subroutine subprogram (see "SUBROUTINE Statement" on page 208)
- A block data subprogram (see "BLOCK DATA Statement" on page 59)
- A common block (see "COMMON Statement" on page 66)
- An external user-supplied subprogram that cannot be classified by its usage in that program unit as either a subroutine or function subprogram name (see "EXTERNAL Statement" on page 95)
- A NAMELIST (see "NAMELIST Statement" on page 148)

Global and Local Names

Due to system restrictions, global names must consist of EBCDIC characters only.

- Classes of global names are:

  Common block
  External function
  Subroutine
  Main program
  Block data subprogram

Local names are recognized internal to the program unit where they are referenced. Local names can consist of EBCDIC characters or of double-byte characters.

- Classes of local names are:

  Array
  Variable
  Constant
  Statement function
  Intrinsic function
  Dummy procedure
  NAMELIST

Names must not be in more than one class within a program unit, except in the following situations:

- A common-block name can also be an array, variable, or statement function name in a program unit.
- A function subprogram name must also be a variable name in the function subprogram.

After a name is used as a main program name, a function subprogram name, a subroutine subprogram name, a block data subprogram name, a common-block name, or an external procedure name in any unit of an executable program, no other program unit of that executable program can use that name to identify an entity of these classes in any other way.

Note: For global names longer than seven characters, the first four and last three characters are concatenated to form the external symbol used to identify the global entity. For example, these names:

```
PASSED_PARAMETER
HASH$FUNCTION
```

would be shortened to the following:

```
PASSTER
HASHION
```

EBCDIC and DBCS Names

Definitions

EBCDIC Name—A sequence of 1 to 6 (31) letters, digits, or underscore characters, the first of which must be a letter.

DBCS Name—A shift-out character (X'0E') followed by a sequence of 1 to 14 double-byte characters terminated by a shift-in character (X'0F'). The sequence must contain at least one double-byte character that does not represent an EBCDIC double-byte character. If the first double-byte character represents an EBCDIC double-byte character, it must be a letter.

An EBCDIC double-byte character must be a letter, digit or underscore in double-byte. Double-byte lowercase letters are equivalent to double-byte uppercase letters.

Valid Symbolic Names:

`X_ray`	(An EBCDIC symbolic name)
`<kk.1.2.3>`	(A DBCS symbolic name)

Source Language Statements

The rules for forming each type of source language statement are defined in Chapter 5, along with a description of that statement's purpose and use. The following discussion of source language statements is limited to the rules by which input lines are classified as comments or other source language statements, and to the correct format of input lines.

There are two major kinds of input lines: statements and comments.

- Statements, which may occupy one or more input lines, provide the information needed by the compiler to create the object program.
- Comments are descriptive remarks about the program unit in which they reside. Comments are copied onto the source program listing. Comments are not present in the object program and have no effect on program execution. Comment lines can be used to separate blocks of source language statements to make the program more readable.

Free-Form Input Format

Free-form input permits greater freedom in arranging the input text of a program than does fixed-form input. The following rules govern free-form input:

- Comments

  A comment line begins with a quotation mark (") or an exclamation point (!) in column 1. This type of comment must not follow a continued line, and cannot itself be continued.

  An in-line comment begins with an exclamation point, which initiates a comment anywhere on a line except when the exclamation point appears within character context. This in-line comment may be interspersed with free-form source lines.

- Statement Text

  The text of free-form statements is entered in up to 80 columns. The first character of a statement (after a label, if any) must be alphabetic. Multiple statements per line are not allowed. Columns 73 through 80 are considered part of the statement text in free form. They may not be used for identification in free-form statements.

  Note for DBCS Representation in Source: The DBCS compiler option must be in effect for double-byte character text to be interpreted correctly; the shift-out/shift-in characters are invalid characters otherwise.

- Statement Labels

  The initial line of a statement may contain a label as the first (leftmost) entry on the line. A label may contain 1 to 5 decimal digits. Blanks and leading zeros are ignored. The value must not be zero. The values of labels do not affect the order in which statements are compiled or executed. Each label must be unique within a program unit.

- Initial Line

  The initial line of a statement may have a label. The first character of the statement text must be alphabetic. If a statement does not have a label, the statement text must begin on the initial line.

- Continued Lines

  The text of any statement, except the END statement, may continue on the following line. A line to be continued is indicated by terminating the line with a minus sign (-).

  When an in-line comment appears on a continued line, the minus sign (-) must precede the exclamation point that begins the comment. If multiple comment lines are used between a continued line and its continuation, each exclamation point (!) must be preceded by a minus sign. See Figure 7 on page 13 for an example.

- Preserving a Minus Sign

  If the last character in a line is a minus sign, the compiler assumes it indicates continuation. If the last two characters in a line are minus signs, only the last one is taken as a continuation character, and the preceding one is preserved as a minus sign.

- Continuation Lines

  A continuation line is a line following a continued line. The statement text may start in any position. In free form source, there is no restriction on the number of continuation lines. A statement is restricted only by the statement length limit.

  Note for DBCS Representation in Source: When a character constant is continued, the shift-in character that immediately precedes the minus sign and the shift-out character on the next line will be ignored (if it is in column 1). In Figure 3, HEAD_LINE1 and HEAD_LINE2 are equivalent:

```
Column:   1                                                      80
          --------------------------------------------. . .------
            CHARACTER*50  HEAD_LINE1
            CHARACTER*50  HEAD_LINE2
              HEAD_LINE1 = '<.A  .H.E.A.D.I.N.G>-
          <  .L.I.N.E>'
              HEAD_LINE2 = '<.A  .H.E.A.D.I.N.G  .L.I.N.E>'
```

Figure 3. Shift-out/Shift-in Characters in Continued Source

- Maximum Statement Length

  The maximum length of a free-form source statement is 6600 characters, excluding the continuation characters and the statement label. Blank characters are counted in the total number of characters. Any blank characters after the continuation characters are not counted.

Figure 4 illustrates free-form source statements.

```
Column:   1
          --------------------
          @PROCESS FREE
          "SAMPLE TEXT
            .
            .
            .
          CHARACTER*200  BLINEO
            IF (IREAD.GE.3) THEN                 !  Get ready for next line
               IF (BLINEO(1:1).NE.'$') THEN
                  IWRITE = IWRITE - 1
                  WRITE (UNIT=7,FMT=200) BLINEO, BLINE1, BLINE2, BLINE3, -
                                         TOKEN1, TOKEN2, TOKEN3, TOKEN4, -
                                         PGMNAME, LOPNAME
               ENDIF
               WRITE (UNIT=6,FMT=300) BLINEO     !  Don't process tokens
             ENDIF
            .
            .
            .
```

Figure 4. Example of Free-Form Source Statements

Fixed-Form Input Format

The statements and comments of a source program in fixed form must conform to the following rules:

- Comments

  A comment line must begin with a C or an asterisk (*) in column 1. In addition, an exclamation point (!) initiates a comment anywhere on a line except when it appears in character context or in column 6 (where it is treated as a continuation character). This in-line comment consists of the exclamation point and all the characters to its right, up to the end of the line. The in-line comment is treated as a blank character.

  Comment lines may appear anywhere in a program unit before the END statement. (Comment lines may precede a continuation line.) Blank lines may appear anywhere in a program unit and are processed as comment lines. Comments may contain double-byte characters delimited by shift-out/shift-in characters.

- Statement Text

  The text of a fixed-form statement is written in columns 7 through 72 on an initial line. The statement text may continue on as many as 99 continuation lines. Multiple statements per line are not allowed. Every statement in a program unit may have a label in columns 1 to 5. Column 6 is used to distinguish between initial and continuation lines. Columns 73 through 80 are not part of the statement and may be used for identification. A statement is terminated by another statement or by the end of the input.

  Note for DBCS Representation in Source: The DBCS compiler option must be in effect for double-byte character text to be interpreted correctly; the shift-out/shift-in characters are invalid characters otherwise.

- Statement Labels

  The statement label consists of from 1 to 5 decimal digits anywhere in columns 1 to 5 in the initial line of a statement. The value must not be zero. Values of labels do not affect the order in which statements are compiled or executed. Each label must be unique within a program unit. Leading zeros are discarded.

- Initial Line

  Column 6 of the initial line of a statement must be a blank or a zero. The initial line of every statement may be labeled. If a statement does not have a label, the statement text must begin on the initial line. The initial line cannot be blank.

- Continuation Lines

  A statement that is not complete on the initial line may continue in columns 7 through 72 on as many as 99 continuation lines. A continuation line must have a character that is not blank or zero in column 6. Columns 1 through 5 on a continuation line may contain characters, but they are ignored.

  Note for DBCS Representation in Source: When a character constant is continued, the shift-in character in column 72 and the shift-out character in column 7 on the next line will be ignored.

In Figure 5, HEAD_LINE1 and HEAD_LINE2 are equivalent:

```
Column:    1    6                                                   72
           ----------------------------. . .----------------------
                 CHARACTER*50 HEAD_LINE1
                 CHARACTER*50 HEAD_LINE2

                 HEAD_LINE1 =                  '<.A  .H.E.A.D.I.N.G>
                X<  .L.I.N.E>'

                 HEAD_LINE2 = '<.A  .H.E.A.D.I.N.G  .L.I.N.E>'
```

Figure 5. Shift-out/Shift-in Characters in Continued Source Lines

- Identification

  Columns 73 through 80 of any input line are not significant to the compiler and may, therefore, be used for identification, sequencing, or any other purpose.

As many blanks as desired may be written on a statement or comment to improve readability. They are ignored by the compiler. However, blanks inserted in literal or character data are retained and treated as blanks within the data.

Figure 6 illustrates fixed-form source statements.

```
Column:  1    6                                73    80
         ---------------------     . . .       --------
         C     SAMPLE TEXT                     SAMP0010
                 .                               .
                 .                               .
                 .                               .
            10 D = 010.5  ! Initialize D       SAMP0210
               GO TO 56                        SAMP0220
           150 A = B + C * (D + E ** F +       SAMP0230
              1  G + H - 2. * (G + P))         SAMP0240
               C = 3.                          SAMP0250
                 .
                 .
                 .
```

Figure 6. Example of Fixed-Form Source Statements

Comments

Comments provide documentation for a program. If you are working with fixed format source code, you must use fixed-form comments; if you are working with free format source code, you must use free-form comments.

A comment may appear anywhere before the END statement.

Free-Form Input

Free-form comments have the following attributes:

- Begins in column 1 with a quotation mark (") in column 1 or with an exclamation point (!) anywhere on a line except within a character or Hollerith constant

- Comments cannot be continued

Valid Comment Placement:

```
Column:  1     7
         ______________________________________________________________
         "THIS COMMENT BEGINS THE PROGRAM
         .
         .
         .
         10D=010.5
         GOTO 56           ! UNCONDITIONAL BRANCH
         150 A=B+C*(D+E**F--  ! A CONTINUED LINE
         -! THIS COMMENT IS VALID, BECAUSE IT IS PRECEDED BY A MINUS SIGN
         G+H-2.*(G+P))
         ! THIS IS ANOTHER COMMENT
         .
         .
         .
         END
```

Fixed-Form Input

Fixed-form comments are indicated by one of the following:

- A "C" or an asterisk (*) appearing in column 1
- An exclamation point (!) appearing anywhere on a line *except* within a character or Hollerith constant or in column 6 (where it acts as a continuation character). This in-line comment consists of the exclamation point and all the characters to its right, up to the end of the line.
- A blank line, which is treated as a comment

Statement Labels

Statement labels uniquely identify statements within a program unit. Labels can be given to every statement; however, a label is significant to the compiler only when it identifies:

- A statement to which control is passed
- The end of a sequence of statements that are to be executed repeatedly
- A formatting statement

A statement label is a sequence of from 1 to 5 decimal digits, one of which must be nonzero. It can be written in either fixed form or free form. See "Statement Labels" on page 207.

Keywords

Keywords identify procedures supplied by VS FORTRAN Version 2 (intrinsic functions), which can be used as part of any program. These procedures are mathematical functions and service subroutines, which are supplied to save programmers' time. See Chapter 5, "Intrinsic Functions" on page 243 and Chapter 7, "Service Subroutines" on page 269.

A keyword is a specified sequence of characters. The context identifies whether a particular sequence of characters is a keyword or a name. There is no keyword that is reserved in all contexts.

Chapter 2. Data

Data is a formal representation of facts, concepts, or instructions. VS FORTRAN Version 2 manipulates three general kinds of data:

- Constants
- Variables
- Arrays

Note: These are not to be confused with *data types.* Data types correspond to the the five types of variables, as discussed under "Data Types and Lengths" on page 22.

Constants

A constant is a fixed, unvarying quantity. There are several classes of constants:

- **Arithmetic** constants specify decimal values. There are three types of arithmetic constants:

  Integer
  Real
  Complex

- **Logical** constants specify a logical value as "true" or "false."
  There are two logical constants:

  .TRUE.
  .FALSE.

- **Character** constants are strings of characters enclosed in apostrophes.

- **Hollerith** constants are used in FORMAT statements, as arguments, and are also accepted in DATA statements as initialization values.

- **Hexadecimal** constants are used only as data initialization values for any type of variable or array.

With the PARAMETER statement, you can give a name to a constant. (See "PARAMETER Statement" on page 158.)

Arithmetic Constants

Arithmetic constants fall into three categories: integer, real, and complex.

An unsigned constant is a constant with no leading sign. A signed constant is a constant with a leading plus or minus sign.

Integer Constants

Definition

Integer Constant—A string of decimal digits containing no decimal point and expressing a whole number.

Maximum Magnitude: 2147483647 (that is, $2^{31} - 1$).

An integer constant may be positive, zero, or negative. If unsigned and nonzero, it is assumed to be positive. (A zero may be written with a preceding sign with no effect on the value.) Its magnitude must not be greater than the maximum, and it must not contain embedded commas. It occupies 4 bytes of storage.

Valid Integer Constants:

```
0

91

173

-2147483647
```

Invalid Integer Constants:

27.	Contains a decimal point.
3145903612	Exceeds the maximum magnitude.
5,396	Contains an embedded comma.
-2147483648	Exceeds the maximum magnitude, even though it fits into 4 bytes.

Real Constants

Definition

Real Constant—A string of decimal digits that expresses a real number. It can have one of three forms: a basic real constant, a basic real constant followed by a real exponent, or an integer constant followed by a real exponent.

Magnitude:	0 or 16^{-65} (approximately 10^{-78}) through 16^{63} (approximately 10^{75})
Precision:	(Four bytes) 6 hexadecimal digits (approximately 6 decimal digits) (Eight bytes) 14 hexadecimal digits (approximately 15 decimal digits) (Sixteen bytes) 28 hexadecimal digits (approximately 32 decimal digits)

A real constant may be positive, zero, or negative (if unsigned and nonzero, it is assumed to be positive) and must be within the allowable range. It may not contain embedded commas. A zero may be written with a preceding sign with no effect on the value. The decimal exponent permits the expression of a real constant as the product of a basic real constant or integer constant and 10 raised to a desired power.

A basic real constant is a string of digits with a decimal point. It is used to approximate the value of the constant in 4 bytes of storage.

The storage requirement (length) of a real constant can be explicitly specified by appending a real exponent to a basic real constant or an integer constant. The valid exponents consist of the letters E, D, or Q, followed by an integer constant.

The letter E specifies a constant of length 4 and occupies 4 bytes of storage; the letter D specifies a constant of length 8 and occupies 8 bytes of storage. The letter Q specifies a constant of length 16 and occupies 16 bytes of storage.

Valid Real Constants (Four Bytes):

`+0.`

`-999.9999`

`7.0E+0` That is, $7.0 \times 10^{0} = 7.0$

`9761.25E+1` That is, $9761.25 \times 10^{1} = 97612.5$

`7.E3`

`7.0E3` That is, $7.0 \times 10^{3} = 7000.0$

`7.0E+03`

`7E-03` That is, $7.0 \times 10^{-3} = 0.007$

`21.98753829457168` Note: This is a valid real constant, but it cannot be accommodated in four bytes. It will be accepted and truncated.

Valid Real Constants (Eight Bytes):

`123456789012345.D-73` Equivalent to $.123456789012345 \times 10^{-58}$

`7.9D03`

`7.9D+03` That is, $7.9 \times 10^{3} = 7900.0$

`7.9D+3`

`7.9D0` That is, $7.9 \times 10^{0} = 7.9$

`7D03` That is, $7.0 \times 10^{3} = 7000.0$

Valid Real Constants (Sixteen Bytes):

`0.234523453456456734565678Q+43`

`5.001Q08`

Invalid Real Constants:

1	Missing a decimal point or a decimal exponent.
3,471.1	Embedded comma.
1.E	Missing a 1- or 2-digit integer constant following the E. It is not interpreted as 1.0 x 10^0.
1.2E+113	Too many digits in the exponent.
23.5D+97	Magnitude outside the allowable range, that is, 23.5 x $10^{97}>16^{63}$.
21.3D-99	Magnitude outside the allowable range, that is, 21.3 x $10^{-99}<16^{-65}$.

Complex Constants

Definition

Complex Constant—An ordered pair of signed or unsigned integer or real constants separated by a comma and enclosed in parentheses. The first constant in a complex constant represents the real part of the complex number; the second represents the imaginary part of the complex number.

The real or integer constants in a complex constant may be positive, zero, or negative and must be within the allowable range. If unsigned and nonzero, they are assumed to be positive. A zero may be written with a preceding sign, with no effect on the value. If both constants are of integer type, however, then both are converted to real type, of 4-byte length. If either constant is of integer type, it is converted to real type. Both constants are converted to the length of the longer constant.

Valid Complex Constants: $(i = \sqrt{-1})$

(3,-1.86)	Has the value 3.0 - 1.86i; both parts are real (4 bytes long).
(-5.0E+03,.16D+02)	Has the value -5000.0 + 16.0i; both parts are double precision.
(4.7D+2,1.973614D4)	Has the value 470.0 + 19736.14i.
(47D+2,38D+3)	Has the value 4700.0 + 38000.i.
(1234.345456567678Q59,-1.0Q-5)	
(45Q6,6E45)	Both parts are real (16 bytes long).

Invalid Complex Constants:

`(A, 3.7)`	Real part is not a constant.
`(.0009Q-1,7643.Q+1199)`	Too many digits in the exponent of the imaginary part.
`(49.76, .015D+92)`	Magnitude of imaginary part is outside of allowable range.

Logical Constants

Definition

Logical Constant—A constant that can have a logical value of either true or false.

There are two logical constants:

.TRUE.
.FALSE.

The words TRUE and FALSE must be preceded and followed by periods. Each occupies 4 bytes.

The logical constant .TRUE. or .FALSE., when assigned to a logical variable, specifies that the value of the logical variable is true or false, respectively. (See "Logical Expressions" on page 40.)

The abbreviations T and F (without the periods) may be used for .TRUE. and .FALSE., respectively, only for the initialization of logical variables or logical arrays in the DATA statement or in the explicit type statement. The following is an example:

```
LOGICAL L1, L2
DATA L1/T/, L2/F/
```

For use as input/output data, see "L Format Code" under "FORMAT Statement."

Character Constants

Definitions

Character Constant—A string of any characters capable of representation in the processor. The string must be enclosed in apostrophes.

A character constant may be composed of EBCDIC and/or double-byte characters. Within character data, the shift-out and shift-in characters are used to delimit double-byte characters.

A character constant may be used as a data initialization value, or in any of the following:

- A character expression
- An assignment statement
- The argument list of a CALL statement or function reference

- An input or output statement
- A FORMAT statement
- A PARAMETER statement
- A PAUSE or STOP statement

The delimiting apostrophes are not part of the data represented by the constant. An apostrophe within the character data is represented by two consecutive apostrophes, with no intervening blanks. In a character constant, blanks embedded between the delimiting apostrophes are significant.

The length of a character constant is the number of character storage units needed to represent the character data between the delimiting apostrophes. (Consecutive apostrophes count as one character storage unit.) The length of a character constant must be greater than zero.

Valid Character Constants:	**Length:**
'DATA'	4
'X-COORDINATE Y-COORDINATE Z-COORDINATE'	38
'3.14'	4
'DON''T'	5
'<.D.O.N.'.T>'	12
'<.D.O.N.'.T>DO'	14
'<ƀƀ>'	4

Hollerith Constants

Definition

Hollerith Constant—A string of any characters capable of representation in the processor and preceded by *w*H, where *w* is the number of characters in the string. The value of *w* (the number of characters in the string), including blanks, may not be less than 1 or greater than 255.

Each character requires one byte of storage.

Hollerith constants can be used in FORMAT statements, as arguments and in initialization statements, other than in CHARACTER initialization.

Valid Hollerith Constants:

```
24H INPUT/OUTPUT AREA NO. 2

6H DON'T
```

Hexadecimal Constants

Definition

Hexadecimal Constant—The character Z, followed by two or more hexadecimal digits formed from the set of characters 0 through 9 and A through F.

A hexadecimal constant may be used as a data initialization value for any type of variable or array.

One byte contains 2 hexadecimal digits. If a constant is specified as an odd number of digits, leading zeros are added on the left to fill the byte. The internal binary form of each hexadecimal digit is as follows:

```
0-0000      4-0100      8-1000      C-1100
1-0001      5-0101      9-1001      D-1101
2-0010      6-0110      A-1010      E-1110
3-0011      7-0111      B-1011      F-1111
```

Valid Hexadecimal Constants:

Z1C49A2F1 represents the bit string:

```
00011100010010011010001011110001
```

ZBADFADE represents the bit string:

```
00001011101011011111101011011110
```

where the first 4 zerò bits are implied in the second bit string because an odd number of hexadecimal digits are written.

The maximum number of digits allowed in a hexadecimal constant depends upon the length specification of the variable being initialized (see "Data Types and Lengths" on page 22). The following list shows the maximum number of digits for each length specification:

Length of Variables	Maximum Number Hexadecimal Digits
32	64
16	32
8	16
4	8
2	4
1	2

If the number of digits is greater than the maximum, the excess leftmost hexadecimal digits are truncated; if the number of digits is less than the maximum, hexadecimal zeros are supplied on the left.

If the variable being initialized is of COMPLEX type, the specification should indicate a single value, rather than a real value and an imaginary value.

Variables

A variable is a data item, identified by a name, that occupies a storage area. (However, in situations involving error or interruption handling, where normal program flow is asynchronously interrupted, a variable may not occupy a storage area.) The value represented by the name is always the current value.

Before a variable has been assigned a value, its content is undefined, and the variable should not be referred to except to assign it a value.

You can set an initial value into a variable using the DATA statement, or the first executable statement that refers to it (for example, a READ statement or an assignment statement) can assign a value to it.

Variable Names

The names of variables are governed by the rules described in "Names" on page 7. The use of meaningful variable names can aid in documenting a program.

Valid Variable Names:

```
XPOSN

$AMOUNT

Z_BOOST

object_class

<kk.1.2.3>
```

Invalid Variable Names:

4ARRAY	First character should be alphabetic.
SI'X	Should not contain a special character.

Data Types and Lengths

The type of a variable corresponds to the type of data the variable represents. (See Figure 7.) Thus, an integer variable must represent integer data, a real variable must represent real data, and so on. There is no data type associated with hexadecimal data; this type of data is identified by a name of one of the other types. There is no data type associated with statement labels; variables that contain the statement label of an executable statement or a FORMAT statement are not considered to contain an integer value. (See "ASSIGN Statement" on page 51.)

For every data type, there is an implicit length specification that determines the number of bytes that are reserved. If you are explicitly defining a noncharacter data type, you may use an optional length specification.

Figure 7 shows each data type with its associated storage length and standard length.

Data Type	Valid Storage Lengths	Default Length
Integer	2 or 4	4
Real	4, 8, or 16	4
Double Precision	8	8
Complex	8, 16, or 32	8
Character	1 through 32767	1
Logical	1 or 4	4

Figure 7. Data Types and Valid Lengths

A programmer may declare the type of variable by using the following:

- Explicit specification statements
- IMPLICIT statement
- Predefined specification contained in the language

An explicit specification statement overrides an IMPLICIT statement, which, in turn, overrides a predefined specification. The optional length specification of a variable may be declared only by explicit or IMPLICIT specification statements. If, in these statements, no length specification is stated, the default length is assumed. INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER are used to specify the length and type in these statements.

VS FORTRAN Version 2 accepts:

- INTEGER*2 to indicate 2 bytes and INTEGER*4 as an alternative to INTEGER, to indicate 4 bytes;
- REAL*4 as an alternative to REAL, to indicate 4 bytes;
- REAL*8 as an alternative to DOUBLE PRECISION, to indicate 8 bytes;
- REAL*16 to indicate 16 bytes;
- LOGICAL*1 to indicate 1 byte;
- LOGICAL*4 to indicate 4 bytes, as an alternative to LOGICAL;
- COMPLEX*8 to indicate 8 bytes, as an alternative to COMPLEX (the first 4 bytes represent a real number and the second 4 bytes represent an imaginary number);
- COMPLEX*16 to indicate 16 bytes (the first 8 bytes represent a real number and the second 8 bytes represent an imaginary number);
- COMPLEX*32 to indicate 32 bytes (the first 16 bytes represent a real number and the second 16 bytes represent an imaginary number).

Type Declaration by Predefined Specification

The predefined specification is a convention used to specify the type and precision of variables as integer or real.

- If the first character of the variable name is I, J, K, L, M, or N, the variable is integer of length 4.

- If the first character of the variable name is any other alphabetic character, the variable is real of length 4.
- If the first character of the variable name is a currency symbol ($), the variable is real of length 4.
- If the tırst double-byte character of a DBCS name represents an EBCDIC double-byte character (that is, if the first byte is X'42'), the three rules above determine the variable's type (based on the second byte).
- If the first double-byte character of a DBCS name represents a character not in the EBCDIC double-byte character set, the variable is real of length 4 and cannot be implicitly typed.

This convention is the traditional FORTRAN method of specifying the type of a variable as either integer or real. Unless otherwise noted, it is assumed in the examples in this publication that this specification applies. Variables defined with this convention are of standard (default) length.

Type Declaration by the IMPLICIT Statement

The IMPLICIT statement allows you to specify the type of variables, in much the same way as the type was specified by the predefined convention. That is, the type is determined by the first character of the variable name. However, by using the IMPLICIT statement, you have the option of specifying which initial characters designate a particular data type. The IMPLICIT statement can be used to specify all types of variables, integer, real, complex, logical, and character, and to indicate storage length.

Note: If the first double-byte character of a DBCS name represents a character not in the EBCDIC double-byte character set, the variable cannot be implicitly typed.

The IMPLICIT statement overrides the data type as determined by the predefined convention.

The IMPLICIT NONE statement voids the implicit type rules for names. When IMPLICIT NONE is specified, all names must be explicitly typed.

The IMPLICIT statement is discussed in "IMPLICIT Statement" on page 133.

Type Declaration by Explicit Specification Statements

Explicit type statements override IMPLICIT statements and predefined specifications. Explicit specification statements are discussed in "Explicit Type Statement" on page 91.

Array

An array is an ordered and structured sequence of data items called array elements. The number and arrangement of elements in an array are specified by the array declarator. The array declarator indicates the number of dimensions and the size of each dimension. An array must have at least one or as many as seven dimensions. A particular element in the array is identified by the array name and its position in the array. All elements of an array have the same type and length.

To refer to any element in an array, the array name plus a parenthesized subscript must be used. In particular, the array name alone does not represent the first element except in an EQUIVALENCE statement.

Before an array element has been assigned a value, its content is undefined and should not be referenced.

You can define an array using the DIMENSION statement, the explicit type statement, or the COMMON statement.

Subscripts

A subscript is a quantity (or a set of subscript expressions separated by commas) associated with an array name to identify a particular element of the array. The number of subscript quantities in any subscript must be the same as the number of dimensions of the array referenced. A subscript is enclosed in parentheses and is written immediately after the array name. A maximum of seven subscript expressions can appear in a subscript.

The following rules apply to the construction of subscripts. (For additional information and restrictions, see Chapter 3, "Expressions" on page 31.)

1. Subscript expressions may contain arithmetic expressions that use any of the arithmetic operators: +, -, *, /, **.
2. Subscript expressions may contain function references that do not change any other value in the same statement.
3. Subscript expressions may contain array elements.
4. Mixed-mode expressions (integer and real only) within a subscript are evaluated according to normal FORTRAN rules. If the evaluated expression is real, it is converted to integer by truncation.
5. The evaluated result of a subscript expression must always be greater than or equal to the corresponding lower dimension bound and must not exceed the corresponding upper dimension bound. (See "Size and Type Declaration of an Array" on page 26.)

Valid Array Elements:

```
ARRAY (IHOLD)

NEXT (19)

MATRIX (I-5)

BAK (I,J(K+2*L,.3*A(M,N)))        J is an array.

ARRAY (I,J/4*K**2)

ARRAY (-5)

LOT (0)
```

Invalid Array Elements:

```
ALL(.TRUE.)          A subscript expression may not be a
                     logical expression.

NXT (1+(1.3,2.0))    A subscript expression may not be a
                     complex expression.
```

Note: The elements of an array are stored in column-major order. To step through the elements of the array in the linearized order defined as "column-major order," each subscript varies (in steps of 1) from its lowest valid value to its highest valid value, such that each subscript expression completes a full cycle before the next subscript expression to the right is increased. Thus, the leftmost subscript expression varies most rapidly, and the rightmost subscript expression varies least rapidly.

The following list is the order of an array named C defined with two dimensions:

```
DIMENSION C(1:3,0:1)

C(1,0)
C(2,0)
C(3,0)
C(1,1)
C(2,1)
C(3,1)
```

Size and Type Declaration of an Array

The size (number of elements) of an array is declared by specifying, in a subscript, the number of dimensions in the array and the size of each dimension. This type of specification is called an "array declarator." Each dimension is represented by an optional lower bound (*e1*) and a required upper bound (*e2*) in the form:

Syntax

name ([*e1*:] *e2*)

name

is an array name.

e1

is the lower dimension bound. It is optional. If *e1* (with its following colon) is not specified, its value is assumed to be 1.

e2

is the upper dimension bound and must always be specified.

The colon represents the range of values for an array's subscript.

The upper and lower bounds (*e1* and *e2*) are arithmetic expressions in which all constants and variables are of integer type.

- The value of the lower bound may be positive, negative, or zero. It is assumed to be 1, if it is not specified.
- A maximum of seven dimensions is permitted. The size of each dimension is equal to the difference between the upper and lower bounds plus 1. If

the value of the lower dimension bound is 1, the size of the dimension is equal to the value of its upper bound.

- Function or array element references are not allowed in dimension bound expressions.
- The value of the upper bound must be greater than or equal to the value of the lower bound. An upper dimension bound of an array, if specified as an asterisk, is always assumed to be greater than or equal to the lower dimension bound.

There are two kinds of arrays:

- An **actual** array is one whose name is not a dummy argument and whose dimension bound expressions can contain only constants or names of constants of integer type.
- A **dummy** array is one whose name must contain a dummy argument and whose dimension bound expressions can also contain:
  - Integer variables that are also dummy arguments
  - Expressions that contain:
    - Signed or unsigned integer constants
    - Names of integer constants
    - Variables that are dummy arguments or appear in a common block in that subprogram

  The upper dimension bound of the last dimension of a dummy array name can be an asterisk. In this case, the dummy array is called an assumed-size array.

Valid Array Declarations:

```
DIMENSION A(0:9),B(3,-2:5)

DIMENSION ARAY(-3:-1),DARY(-3:ID3**ID1)

DIMENSION IARY(3)
```

Size information must be given for all actual arrays in a program, so that an appropriate amount of storage may be reserved. Declaration of this information is made by one of the following:

- a DIMENSION statement (see "DIMENSION Statement" on page 76)
- a COMMON statement (see "COMMON Statement" on page 66)
- an explicit type statement (see "Explicit Type Statement" on page 91)

The type of an array name is determined by the conventions for specifying the type of a variable name. Each element of an array is of the type and length specified for the array name.

Object-Time Dimensions

If a dummy argument array is used in a function or subroutine subprogram, the absolute dimensions of the array do not have to be explicitly declared in the subprogram by constants. Instead, the array declarators appearing in an explicit specification statement or DIMENSION statement in the subprogram may contain dummy arguments or variables in the common block that are integer variables of length 4, to specify the size of the array.

When the subprogram is called, these integer variables receive their values from the actual arguments in the calling program reference or from the common block. As a result, the dimensions of a dummy array appearing in a subprogram may change each time the subprogram is called. This is called an "adjustable array" or an "object-time dimension array."

The absolute dimensions of an array must be declared in the calling program or in a higher level calling program, and the array name must be passed to the subprogram in the argument list of the calling program. The dimensions passed to the subprogram must be less than or equal to the absolute dimensions of the array declared in the calling program. The variable dimension size can be passed through more than one level of subprogram (that is, to a subprogram that calls another subprogram, passing it dimension information).

Integer variables in the explicit specification or DIMENSION statement that provide dimension information may be redefined within the subprogram, but the redefinitions have no effect on the size of the array. The size of the array is determined at the entry point at which the array information is passed.

Character arrays are specified in the same manner as other data types. (See "DIMENSION Statement" on page 76 and "Explicit Type Statement" on page 91.) The length of each array element is either the standard length of 1 or may be declared larger with a type or IMPLICIT statement. Each character array element is treated as a single entity. Portions of an array element can be accessed through substring notation.

Character Substrings

A character substring is a contiguous portion of a character variable or character array element. A character substring is identified by a substring reference. It may be assigned values and may be referred to. A substring reference is local to a program unit.

The form of a substring reference is:

Syntax

a(*e1*:*e2*)

a

is a character variable name or a subscripted character array name (see "Array" on page 24).

e1 and *e2*

are substring expressions.

Substring expressions are optional, but the colon (:) is always required inside the parentheses. The colon represents a range of values. If *e1* is omitted, a value of one is implied for *e1*. If *e2* is omitted, a value equal to the length of the character variable or array element is implied for *e2*. Both *e1* and *e2* may be omitted; for example, the form *v*(:) is equivalent to *v*.

The value of *e1* specifies the leftmost character position and the value of *e2* specifies the rightmost character position of the substring. The substring information (if any) must be specified after the subscript information (if any).

- The values of *e1* and *e2* must be integer, positive, and nonzero.
- The value of *e1* must be less than or equal to the value of *e2*.
- The values of *e1* and *e2* must be less than or equal to the number of characters contained in the corresponding variable name or array element.

Note for Double-Byte Characters: Since double-byte characters take two bytes of storage, using substring references will give unpredictable results. Substring references could split up the characters or cause unbalanced shift codes (see Example 3). Therefore, the ASSIGNM subroutine should be used for string operations on double-byte characters. For more information, see "ASSIGNM Subroutine" on page 277.

Example 1:

Given the following statements:

```
CHARACTER*5 CH(10)
CH(2)='ABCDE'
```

then

```
CH(2)(1:2) has the value AB.
CH(2)(:3)  has the value ABC.
CH(2)(3:)  has the value CDE.
```

Example 2:

Given the following statements:

```
CHARACTER*5 SUBSTG, SYMNAM
SYMNAM= 'VWXYZ'
I = 3
J = 4
SUBSTG(1:2) = SYMNAM(I:J)
SUBSTG(I:J) = SYMNAM(1:2)
SUBSTG(J+1:) = SYMNAM(5:)
```

then SUBSTG has the value XYVWZ.

Example 3:

Substring operations on double-byte character data may split up a shift-out/shift-in pair. Using such a substring will give unpredictable results.

Given the following statements:

```
CHARACTER LOGO*8, FRONT*4
LOGO = '<.I.B.M>'
FRONT = LOGO(1:4)
```

then FRONT has the value <.I. which contains unbalanced shift codes.

The ASSIGNM subroutine should be used for string operations on double-byte characters. For more information, see "ASSIGNM Subroutine" on page 277.

Chapter 3. Expressions

There are four kinds of expressions: arithmetic, character, relational, and logical.

- The value of an arithmetic expression is always a number whose type is integer, real, or complex.
- The value of a character expression is a character string.
- The value of a relational or logical expression is always a .TRUE. or .FALSE. logical value.

Evaluation of Expressions

Expressions are evaluated according to the following rules:

- Any variable, array element, function, or character substring referred to as an operand in an expression must be defined (that is, must have been assigned a value) at the time the reference is executed.

  In an expression, an integer operand must be defined with an integer value, rather than a statement label. (See "ASSIGN Statement" on page 51.) If a character string or a substring is referred to, all the characters referred to must be defined at the time the reference is executed.

- The execution of a function reference in a statement must not alter the value of any other entity within the statement in which the function reference appears. Also, it must not alter the value of any entity in the common block that affects the value of any other function reference in that statement.

  If a function reference in a statement alters the value of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the statement. For example, the following statements are prohibited if the reference to the function F defines I or if the reference to the function G defines X:

  ```
  A(I) = F(I)

  Y = G(X) + X
  ```

  The data type of an expression in which a function reference appears does not affect the evaluation of the actual arguments of the function.

- An argument to a statement function reference must not be altered by the evaluation of that reference.
- Any array element reference requires the evaluation of its subscript. The data type of an expression in which an array reference appears does not affect, nor is it affected by, the evaluation of the subscript.
- Any execution of a substring reference requires the evaluation of its substring expressions. The data type of an expression in which a substring name appears does not affect, nor is it affected by, the evaluation of the substring expressions.

Arithmetic Expressions

The simplest arithmetic expression consists of a single primary, which may be a constant, name of a constant, variable, array element, function reference, or another expression enclosed in parentheses. The type of an arithmetic expression must be integer, real, or complex.

In an expression consisting of a single primary, the type of the primary is the type of the expression. Examples of arithmetic expressions are shown in Figure 8.

Primary	Type of Primary	Type	Length
3	Integer constant	Integer	4
A	Real variable	Real	4
3.14E3	Real constant	Real	4
3.14D3	Real constant	Double precision	8
(2.0,5.7)	Complex constant	Complex	8
SIN(X)	Real function reference	Real	4
(A*B+C)	Parenthesized real expression	Real	4

Figure 8. Examples of Arithmetic Expressions

Arithmetic Operators

More complicated arithmetic expressions containing two or more primaries may be formed by using arithmetic operators that express the computation(s) to be performed.

The arithmetic operators are shown in Figure 9.

Arithmetic Operator	Definition
**	Exponentiation
*	Multiplication
/	Division
+	Addition (or unary plus)
−	Subtraction (or unary minus)

Figure 9. Arithmetic Operators

Rules for Constructing Arithmetic Expressions

The following are the rules for constructing arithmetic expressions that contain arithmetic operators:

- All desired computations must be specified explicitly. That is, if more than one primary appears in an arithmetic expression, they must be separated from one another by an arithmetic operator. For example, the two variables A and B are not multiplied if written:

  ```
  AB
  ```

  In fact, AB is regarded as a single variable with a two-letter name.

If multiplication is desired, the expression must be written as follows:

```
A*B or B*A
```

- No two arithmetic operators may appear consecutively in the same expression. For example, the following expressions are invalid:

  ```
  A*/B and A*-B
  ```

  The expression A*-B could be written correctly as

  ```
  A*(-B)
  ```

  Two asterisks (**) designate exponentiation, not two multiplication operations.

- Order of Computation

  When expressions are evaluated, operations are examined from left to right, comparing successive operators. Successive operators are evaluated according to the hierarchy shown in Figure 10.

Operation	Hierarchy
Evaluation of functions	1st
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th

Figure 10. Hierarchy of Arithmetic Operations

Note: A unary plus or minus has the same hierarchy as a plus or minus in addition or subtraction.

Successive operators are evaluated left to right based on the hierarchy of operations. If two or more operators of the same priority appear successively in the expression, the order of priority of those operators is from left to right, except for successive exponentiation operators, where the evaluation is from right to left.

Consider the evaluation of the expression in the assignment statement:

```
RESULT=A*B+C*D**I
```

```
1. A*B     Call the result X (multiplication)    (X+C*D**I)
2. D**I    Call the result Y (exponentiation)    (X+C*Y)
3. C*Y     Call the result Z (multiplication)    (X+Z)
4. X+Z     Final operation (addition)
```

The expression:

```
A**B**C
```

is evaluated as follows:

```
1. B**C   Call the result Z
2. A**Z   Final operation
```

Expressions with a unary minus are treated as follows:

```
A=-B is treated as A=0-B

A=-B*C is treated as A=-(B*C)    Because * has higher precedence
                                 than -

A=-B+C is treated as A=(-B)+C    Because - has equal precedence
                                 to +
```

Use of Parentheses in Arithmetic Expressions

Because the order of evaluation (and, consequently, the result) of an expression can be changed through the use of parentheses, refer to Figure 11, Figure 12, and Figure 13 to determine the type and length of intermediate results. Where parentheses are used, the expression contained within the most deeply nested parentheses (that is, the innermost pair of parentheses) is evaluated first. A parenthesized expression is considered a primary.

For example, the expression,

```
B/((A-B)*C)+A**2
```

is effectively evaluated in the following order:

```
1. A-B     Call the result W     B/(W*C)+A**2
2. W*C     Call the result X     B/X+A**2
3. B/X     Call the result Y     Y+A**2
4. A**2    Call the result Z     Y+Z
5. Y+Z     Final operation
```

Type and Length of the Result of Arithmetic Expressions

The type and length of the result of an operation depend upon the type and length of the two operands (primaries) involved in the operation.

Note: Except for a value raised to an integer power, if two operands are of different type and length, the operand that differs from the type and/or length of the result is converted to the type and/or length of the result. Thus the operator operates on a pair of operands of matching type and length.

A negative operand (either real or integer) may not have a real exponent.

When an operand of real or complex type is raised to an integer power, the integer operand is not converted. The resulting type and length match the type and length of the base.

Figure 11 shows the type and length of the result of adding, subtracting, multiplying, or dividing when the first operand is an **integer**.

First Operand	Second Operand	Result
Integer (2)	Integer (2)	Integer (2)
	Integer (4)	Integer (4)
	Real (4)	Real (4)
	Real (8)	Real (8)
	Real (16)	Real (16)
	Complex (8)	Complex (8)
	Complex (16)	Complex (16)
	Complex (32)	Complex (32)
Integer (4)	Integer (2)	Integer (4)
	Integer (4)	Integer (4)
	Real (4)	Real (4)
	Real (8)	Real (8)
	Real (16)	Real (16)
	Complex (8)	Complex (8)
	Complex (16)	Complex (16)
	Complex (32)	Complex (32)

Figure 11. Type and Length of Result Where the First Operand is Integer

Figure 12 shows the type and length of the result of adding, subtracting, multiplying, or dividing when the first operand is **real**.

First Operand	Second Operand	Result
Real (4)	Integer (2)	Real (4)
	Integer (4)	Real (4)
	Real (4)	Real (4)
	Real (8)	Real (8)
	Real (16)	Real (16)
	Complex (8)	Complex (8)
	Complex (16)	Complex (16)
	Complex (32)	Complex (32)
Real (8)	Integer (2)	Real (8)
	Integer (4)	Real (8)
	Real (4)	Real (8)
	Real (8)	Real (8)
	Real (16)	Real (16)
	Complex (8)	Complex (16)
	Complex (16)	Complex (16)
	Complex (32)	Complex (32)
Real (16)	Integer (2)	Real (16)
	Integer (4)	Real (16)
	Real (4)	Real (16)
	Real (8)	Real (16)
	Real (16)	Real (16)
	Complex (8)	Complex (32)
	Complex (16)	Complex (32)
	Complex (32)	Complex (32)

Figure 12. Type and Length of Result Where the First Operand Is Real

Figure 13 shows the type and length of the result of adding, subtracting, multiplying, or dividing when the first operand is **complex**.

First Operand	Second Operand	Result
Complex (8)	Integer (2)	Complex (8)
	Integer (4)	Complex (8)
	Real (4)	Complex (8)
	Real (8)	Complex (16)
	Real (16)	Complex (32)
	Complex (8)	Complex (8)
	Complex (16)	Complex (16)
	Complex (32)	Complex (32)
Complex (16)	Integer (2)	Complex (16)
	Integer (4)	Complex (16)
	Real (4)	Complex (16)
	Real (8)	Complex (16)
	Real (16)	Complex (32)
	Complex (8)	Complex (16)
	Complex (16)	Complex (16)
	Complex (32)	Complex (32)
Complex (32)	Integer (2)	Complex (32)
	Integer (4)	Complex (32)
	Real (4)	Complex (32)
	Real (8)	Complex (32)
	Real (16)	Complex (32)
	Complex (8)	Complex (32)
	Complex (16)	Complex (32)
	Complex (32)	Complex (32)

Figure 13. Type and Length of Result Where the First Operand Is Complex

Examples of Arithmetic Expressions

Assume that the type of the following variables has been specified as indicated below:

Name	Variable Type	Length
I, J, K	Integer	4, 2, 2
C	Real	4
D	Complex	16

Then the expression I*J/C**K+D is evaluated as follows:

Subexpression	Type and Length
I*J (Call the result M)	Integer of length 4
C**K (Call the result Y)	Real of length 4
M/Y (Call the result Z)	Real of length 4
Z+D	Complex of length 16

Note: M is converted to real of length 4 before division is performed.

Z is expanded to the real variable of length 8, and a complex quantity of length 16 (call it W) is formed, in which the real part is the expansion of Z and the imaginary part is zero. The real part of W is then added to the real part of D, and the imaginary part of W is added to the imaginary part of D.

Thus, the final type of the entire expression is complex of length 16, but the types of the intermediate expressions change at different stages in the evaluation.

Depending on the values of the variables involved, the result of the expression I*J*C might be different from I*C*J. This may occur because of the number of conversions performed during the evaluation of the expression.

Because the operators are the same, the order of the evaluation is from left to right. With I*J*C, a multiplication of the two integers I*J yields an intermediate result of integer type and length 4. This intermediate result is converted to a real type of length 4, and multiplied with C of real type of length 4, to yield a real type of length 4 result.

With I*C*J, the integer I is converted to a real type of length 4, and the result is multiplied with C of real type of length 4, to yield an intermediate result of real type of length 4. The integer J is converted to a real type of length 4, and the result is multiplied with the intermediate result to yield a real type of length 4 result.

Evaluation of I*J*C requires one conversion and I*C*J requires two conversions. The expressions require that the computation be performed with different types of arithmetic. This may yield different results.

When division is performed using two integers, any remainder is truncated (without rounding) and an integer quotient is given. If the mathematical quotient is less than 1, the answer is 0. The sign is determined according to the rules of algebra. For example:

I	J	I/J
9	2	4
5	-2	-2
1	-4	0

Character Expressions

The simplest form of a character expression is a character constant, a character variable reference, a character array element reference, a character substring reference, or a character function reference. More complicated character expressions may be formed by using one or more character operands, together with character operators and parentheses.

The character operator is shown in Figure 14.

Character Operator	Definition
//	Concatenation

Figure 14. Character Operator

The concatenation operation joins the operands in such a way that the last character of the operand to the left immediately precedes the first character of the operand to the right.

For example:

```
'AB'//'CD' yields the value of 'ABCD'
```

and

```
'<.W.X>'//'<.Y.Z>' yields the value of '<.W.X><.Y.Z>'
```

The result of a concatenation operation is a character string consisting of the values of the operands concatenated left to right, and its length is equal to the sum of the lengths of the operands.

Note: Except in a CHARACTER assignment statement, the operands of a concatenation operation must not have inherited length. That is, their length specification must not be an asterisk (*) unless the operand is the name of a constant. See "Explicit Type Statement" on page 91.

Use of Parentheses in Character Expressions

Parentheses have no effect on the value of a character expression. For example, if X has the value 'AB', Y has the value 'CD', and Z has the value 'EF',

then the two expressions:

```
X//Y//Z
```

```
X//(Y//Z)
```

both yield the same result, the value 'ABCDEF'.

Relational Expressions

Relational expressions are formed by combining two arithmetic expressions with a relational operator, or two character expressions with a relational operator.

The six relational operators are shown in Figure 15.

Relational Operator	Definition
.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to

Figure 15. Relational Operators

Relational operators:

- Express a condition that can be either true or false.
- May be used to compare two arithmetic expressions (except complex) or two character expressions. Only the .EQ. and .NE. operators may be used to compare an arithmetic expression with a complex expression. If the two arithmetic expressions being compared are not of the same type or length,

they are converted following the rules indicated in Figure 11, Figure 12, and Figure 13.

- Are not allowed in comparisons of arithmetic expressions to character expressions or vice versa.

In the case of character expressions, the shorter operand is considered as being extended temporarily on the right with blanks to the length of the longer operand. The comparison is made from left to right, character by character, according to the collating sequence, as shown in Figure 2 on page 6.

Note: The comparison for double-byte characters is made from left to right, byte by byte, according to the binary value of double-byte code.

Examples:

Assume that the type of the following variables has been specified as indicated:

Variable Names	Type
ROOT, E	Real
A, I, F	Integer
L	Logical
C	Complex
CHAR	Character of length 10

Then the following examples illustrate valid and invalid relational expressions.

Valid Relational Expressions:

```
E .LT. I

E**2.7 .LE. (5*ROOT+4)

.5 .GE. (.9*ROOT)

E .EQ. 27.3E+05

CHAR .EQ. 'ABCDEFGH'

C.NE. CMPLX(ROOT,E)
```

Invalid Relational Expressions:

```
C.GE.(2.7,5.9E3)    Complex quantities can be compared only
                    for equal or not equal in relational
                    expressions.

L.EQ.(A+F)          Logical quantities may never be compared by
                    relational operators.

E**2 .LT 97.1E1     There is a missing period immediately
                    after the relational operator.

 .GT.9              There is a missing arithmetic expression
                    before the relational operator.

E*2  .EQ. 'ABC'     A character expression may not be compared
                    with an arithmetic expression.
```

Length of a Relational Expression:

A relational expression is always evaluated to a LOGICAL*4 result, but the result can be converted in an assignment statement to LOGICAL*1.

Logical Expressions

The simplest form of logical expression consists of a single logical primary. A logical primary can be a logical constant, a name of a logical constant, a logical variable, a logical array element, a logical function reference, a relational expression (which may be an arithmetic relational expression or a character relational expression), or a logical expression enclosed in parentheses. A logical primary, when evaluated, always has a value of true or false.

More complicated logical expressions may be formed by using logical operators to combine logical primaries.

Logical Operators

The logical operators are shown in Figure 16. (A and B represent logical constants or variables, or expressions containing relational operators.)

Logical Operator	Use	Meaning
.NOT.	.NOT.A	If A is true, then .NOT.A is false; if A is false, then .NOT.A is true.
.AND.	A.AND.B	If A and B are both true, then A.AND.B is true; if either A or B or both are false, then A.AND.B is false.
.OR.	A.OR.B	If either A or B or both are true, then A.OR.B is true; if both A and B are false, then A.OR.B is false.
.EQV.	A.EQV.B	If A and B are both true or both false, then A.EQV.B is true; otherwise, it is false.
.NEQV.	A.NEQV.B	If A and B are both true or both false, then A.NEQV.B is false; otherwise, it is true.

Figure 16. Logical Operators

The only valid sequences of two logical operators are:

.AND..NOT.

.OR..NOT.

.EQV..NOT.

.NEQV..NOT.

The sequence .NOT..NOT. is invalid.

Only those expressions that have a value of true or false when evaluated, may be combined with the logical operators to form logical expressions.

Examples:

Assume that the types of the following variables have been specified as indicated:

Variable Names	**Type**
ROOT, E	Real
A, I, F	Integer
L, W	Logical
CHAR, SYMBOL	Character of lengths 3 and 6, respectively

Then the following examples illustrate valid and invalid logical expressions using both logical and relational operators.

Valid Logical Expressions:

```
(ROOT*A .GT. A) .AND. W

L .AND. .NOT. (I .GT. F)

(E+5.9E2 .GT. 2*E) .OR. L

.NOT. W .AND. .NOT. L

L .AND. .NOT. W .OR. CHAR//'123'.LT.SYMBOL

(A**F .GT. ROOT .AND. .NOT. I .EQ. E)
```

Invalid Logical Expressions:

```
A.AND.L         A is not a logical expression.

.OR.W           .OR. must be preceded by a logical
                expression.

NOT.(A.GT.F)    There is a missing period before the logical
                operator .NOT..

L.AND..OR.W     The logical operators .AND. and .OR. must
                always be separated by a logical expression.

.AND.L          .AND. must be preceded by a logical
                expression.
```

Order of Computations in Logical Expressions

In the evaluation of logical expressions, priority of operations involving *arithmetic* operators is as shown in Figure 17.
Within a hierarchic level, computation is performed from left to right.

Operation Involving Arithmetic Operators	Hierarchy
Evaluation of functions	1st (highest)
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th
Relationals (.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.)	5th
.NOT.	6th
.AND.	7th
.OR.	8th
.EQV. or .NEQV.	9th

Figure 17. Hierarchy of Operations Involving Arithmetic Operators

In the evaluation of logical expressions, priority of operations involving *character* operators is as shown in Figure 18. Within a hierarchic level, computation is performed from left to right.

Operation Involving Character Operators	Hierarchy
Evaluation of functions	1st (highest)
Concatenation (//)	2nd
Relationals (.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.)	3th
.NOT.	4th
.AND.	5th
.OR.	6th
.EQV. or .NEQV.	7th

Figure 18. Hierarchy of Operations Involving Character Operators

Example:

Assume the type of the following variables has been specified as follows:

Variable Names	Type	Length
B,D	REAL	4
A	REAL	8
L,N	LOGICAL	4

The expression

```
A.GT.D**B.AND..NOT.L.OR.N
```

is effectively evaluated in the following order (and from left to right):

1. D**B Call the result W.

   Exponentiation is performed because arithmetic operators have a higher priority than relational operators, yielding a real result W of length 4.

2. A.GT.W Call the result X.

The real variable A of length 8 is compared with the real variable W, which was extended to a length of 8, yielding a logical result X, whose value is either true or false.

3. .NOT.L Call the result Y.

   The logical negation is performed because .NOT. has a higher priority than .AND., yielding a logical result Y, whose value is true if L is false and false if L is true.

4. X.AND.Y Call the result Z.

   The logical operator .AND. is applied because .AND. has a higher priority than .OR., to yield a logical result Z, whose value is true if both X and Y are true and false, if both X and Y are false, or if either X or Y is false.

5. Z.OR.N

   The logical operator .OR. is applied to yield a logical result of true if either Z or N is true or if both Z and N are true. If both Z and N are false, the logical result is false.

Note: Calculating the value of logical expressions may not always require that all parts be evaluated. Functions within logical expressions may or may not be invoked. For example, assume a logical function called LGF. In the expression A.OR.LGF(.TRUE.), it should not be assumed that the LGF function is always invoked, because it is not always necessary to do so to evaluate the expression when A is true.

Use of Parentheses in Logical Expressions

Parentheses may be used in logical expressions to specify the order in which the operations are to be performed. Where parentheses are used, the expression contained within the innermost pair of parentheses is evaluated first.

Example:

Assume the type of the following variables specified as follows:

Variable Names	Type	Length
B	REAL	4
C	REAL	8
K, L	LOGICAL	4

The expression

```
.NOT.((B.GT.C.OR.K).AND.L)
```

is evaluated in the following order:

1. B.GT.C Call the result X.

   B is extended to a real variable of length 8 and compared with C of length 8, yielding a logical result X of length 4, whose value is true if B is greater than C or false if B is less than or equal to C.

2. X.OR.K Call the result Y.

   The logical operator .OR. is applied to yield a logical result of Y, whose value is true if either X or K is true or if both X and K are true. If both X and K are false, the logical result Y is false.

3. Y.AND.L   Call the result Z.

   The logical operator .AND. is applied to yield a logical result Z, whose value is true if both Y and L are true and false if both Y and L are false or if either Y or L is false.

4. .NOT.Z

   The logical negation is performed to yield a logical result, whose value is true if Z is false and false if Z is true.

If it contains two or more quantities, a logical expression to which the logical operator .NOT. applies must be enclosed in parentheses. Otherwise, because of the higher precedence of the .NOT. operator, it will apply to the first operand of the relation. For example, assume that the values of the logical variables, A and B, are false and true, respectively. Then the following two expressions are not equivalent:

```
.NOT.(A.OR.B)

.NOT.A.OR.B
```

In the first expression, A.OR.B is evaluated first. The result is true; but .NOT.(.TRUE.) is the equivalent of .FALSE.. Therefore, the value of the first expression is false.

In the second expression, .NOT.A is evaluated first. The result is true; but .TRUE..OR.B is the equivalent of .TRUE.. Therefore, the value of the second expression is true.

The lengths of the results of the various logical operations are shown in Figure 19. (The result of logical operations is always logical of length 4.)

First Operand	Second Operand	Result
Logical (1)	Logical (1) Logical (4)	Logical (4) Logical (4)
Logical (4)	Logical (1) Logical (4)	Logical (4) Logical (4)

Figure 19. Type and Length of the Result of Logical Operations

Chapter 4. Statements

A source program consists of a set of statements from which the compiler generates machine instructions and allocates storage for data areas. A statement performs one of three functions:

- It performs certain executable operations (for example, addition, multiplication, branching).
- It specifies the nature of the data being handled.
- It specifies the characteristics of the source program.

Statements are either executable or nonexecutable.

Statement Categories

Statements are divided into the following categories according to what they do:

- Assignment statements
- Control statements
- DATA statement
- Debug statements
- Input/output statements
- PROGRAM statement
- Specification statements
- Subprogram statements
- Compiler directives

Assignment Statements

There are four types of assignment statements: the arithmetic, character, and logical assignment statements and the ASSIGN statement. Execution of an assignment statement assigns a value to a variable or array element.

Control Statements

In the absence of control statements, statements are executed sequentially. Control statements can alter this normal sequence of execution of statements in the program. Control statements are executable. The following are control statements:

CALL
CONTINUE
DO
DO WHILE
END
END DO
GO TO
IF (ELSE, ELSE IF, END IF)
PAUSE
RETURN
STOP

DATA Statement

The DATA statement assigns initial values to variables, array elements, arrays, and substrings. It is nonexecutable.

Debug Statements

The static debug facility is a programming aid that can help locate errors in a source program. This facility traces the flow of execution within a program, displays the values of variables and arrays, and checks the validity of subscripts. DISPLAY, TRACE OFF, and TRACE ON are executable; AT, DEBUG, and END DEBUG are nonexecutable. The following are debug statements:

AT
DEBUG
DISPLAY
END DEBUG
TRACE OFF
TRACE ON

Do not confuse the static debug with Interactive Debug, described in *VS FORTRAN Version 2 Interactive Debug Guide and Reference.* Interactive Debug provides more function and is preferable for debugging.

Input/Output Statements

Input/output (I/O) statements transfer data between two areas of internal storage or between internal storage and an input/output device. Examples of input/output devices are card readers, printers, punches, magnetic tapes, disk storage units, and terminals.

I/O statements allow the programmer to specify how to process the files at different times during the execution of a program. Except for the FORMAT statements, these statements are executable. The following are input/output statements:

BACKSPACE
CLOSE
DELETE
ENDFILE
FORMAT
INQUIRE
OPEN
PRINT
READ
REWIND
REWRITE
WAIT
WRITE

Note: The description of the input and output statements is made from the point of view of a VS FORTRAN Version 2 program. Therefore, words such as *file*, *record*, or *OPEN* must not be confused with the same words used when discussing an operating system. (See the description of each I/O statement later in this chapter.)

Input/Output Semantics

The VS FORTRAN input/output statements are based on a set of semantics which govern file naming, file and unit existence, and file/unit connection. The following discussion provides an introduction to the I/O semantics; for a more in-depth discussion, see *VS FORTRAN Version 2 Programming Guide.*

FORTRAN recognizes both unnamed and named files. In VS FORTRAN programs, all files are referred to by their *ddnames*; a ddname is a name that identifies an operating system file definition which in turn refers to an actual file. Certain ddnames are reserved by VS FORTRAN for use in referring to an unnamed file. Files that are referred to by the reserved ddnames cannot be specified by name in a VS FORTRAN program. For this reason, these files are called unnamed files. (Naming conventions for the reserved ddnames are listed on page 139.) A named file is specified in a VS FORTRAN program by its ddname or by its CMS file identifier or MVS data set name. The ddname for a named file may not be one of the reserved ddnames.

With dynamic file allocation, the user does not need to supply an explicit file definition for a file to be connected to a unit. VS FORTRAN will create the file definition. See *VS FORTRAN Version 2 Programming Guide* for more information on dynamic file allocation.

In FORTRAN, files and units are either existent or non-existent. On a conceptual level, a file exists if it actually resides on the medium and an operating system file definition statement is in effect for the file.

Note: File existence properties differ depending on the type of I/O device being used. See *VS FORTRAN Version 2 Programming Guide*, for specific information on devices and file existence.

A unit is a means of referring to a file so that the file can be used in an input/output operation. Units are referred to in VS FORTRAN programs by a unit identifier. A unit is considered to exist if the unit identifier is valid for your installation. (If you are unsure what the valid unit identifiers are for your installation, see your system programmer.)

Before any data can be transferred, a file must be connected to a unit. In FORTRAN, a unit is a means of accessing a file. Units and files become connected through an OPEN statement, or through preconnection. An OPEN statement associates a file with a unit.

Preconnected files are files that, at the beginning of program execution, are connected to units; that is, the VS FORTRAN program does not need to explicitly associate a file with a unit using an OPEN statement. Only unnamed files may be preconnected.

Once a file is no longer needed by a VS FORTRAN program, it can be disconnected from a unit. Files are disconnected through a CLOSE statement, at the end of program execution, or through an implicit close operation. An implicit close operation occurs when an OPEN statement is issued for a file that is different from the file already connected to the unit.

At the time a file is disconnected, the file may also be deleted. When a file is deleted, VS FORTRAN considers that the file no longer exists. For details on the circumstances under which a file will be deleted, see *VS FORTRAN Version 2 Programming Guide*.

Files that have been disconnected from a unit may be reconnected only by an OPEN statement. Under certain circumstances, I/O statements other than OPEN, CLOSE, and INQUIRE may be issued for a disconnected unit; see *VS FORTRAN Version 2 Programming Guide* for details.

PROGRAM Statement

The PROGRAM statement can be used only for naming a main program; however, it is not required. The PROGRAM statement is nonexecutable.

Specification Statements

Specification statements provide the compiler with information about the nature of the data in the source program. In addition, they supply the information required to allocate storage for this data.

If used, the specification statements must follow the PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statement. They may be preceded by a FORMAT or an ENTRY statement. Specification statements are nonexecutable. The following are specification statements:

COMMON
DIMENSION
EQUIVALENCE
EXTERNAL
IMPLICIT
INTRINSIC
NAMELIST
PARAMETER
SAVE

Explicit type:

CHARACTER
COMPLEX
DOUBLE PRECISION
INTEGER
LOGICAL
REAL

Subprogram Statements

There are three subprogram statements: FUNCTION, SUBROUTINE, and BLOCK DATA.

The function subprogram begins with a FUNCTION statement. At least one executable statement in the function must assign a result to the function name. This value is returned to the calling program unit as the result of the function. The function subprogram is processed whenever its name is appropriately referenced in another program unit.

Subroutine subprograms begin with the SUBROUTINE statement. Like the function subprogram, the subroutine can be a set of commonly used computations, but it need not return any result to the calling program. The subroutine is processed whenever its name is referenced in a CALL statement.

The BLOCK DATA statement begins a block data subprogram. A block data subprogram is used to initialize values for variables and array elements in named common blocks.

Subprogram statements are nonexecutable.

Compiler Directives

EJECT and INCLUDE are IBM extensions that direct the compiler to respectively start a new page or insert one or more source statements into the program. For details on vector directives, see *VS FORTRAN Version 2 Programming Guide*.

Order of Statements in a Program Unit

The order of statements in a program unit (other than a BLOCK DATA subprogram) is as follows:

1. PROGRAM or subprogram statement, if any.
2. IMPLICIT statements, if any.
3. Other specification statements, if any. (Explicit specification statements that initialize variables or arrays must follow other specification statements that contain the same variable or array names.)
4. For the order of data statements, see Figure 20 on page 50.
5. Statement function definitions, if any.
6. Executable statements.
7. END statement.

For the order of DEBUG statements, see "DEBUG Statement" on page 71.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements. Any specification statement that specifies the type of named constant must precede the PARAMETER statement that defines that particular named constant; the named constants referenced in a PARAMETER statement must have been defined by preceding PARAMETER statements.

FORMAT and ENTRY statements may appear anywhere after the PROGRAM or subprogram statement and before the END statement. The ENTRY statement, however, may not appear between a block IF statement and its corresponding END IF statement or within the range of a DO. DATA statements must follow the IMPLICIT statements but may be intermixed with the other specification statements. DATA statements must follow all other specification statements which refer to the same item (variable or array).

A NAMELIST statement declaring a NAMELIST name must precede the use of that name in any input/output statement. Its placement is as indicated for other specification statements.

The order of statements in BLOCK DATA subprograms is discussed under "BLOCK DATA Statement" on page 59.

Figure 20 shows the order of statements.

- The vertical lines in the figure delineate varieties of statements that may be interspersed. For example, FORMAT statements may be interspersed with statement function statements and executable statements.
- Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements.

<table>
<tr><td rowspan="5">Comment Lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY Statements</td><td>IMPLICIT NONE Statement</td><td>IMPLICIT Statements</td></tr>
<tr><td rowspan="3">DATA Statements</td><td>Other Specification Statements</td></tr>
<tr><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="4">END Statement</td></tr>
</table>

Figure 20. Order of Statements and Comment Lines

Statement Descriptions

The rules for coding each statement are described in this section, in alphabetic sequence. Examples are included. For additional examples and explanations, see *VS FORTRAN Version 2 Programming Guide.*

Notes:

1. Comments and statement labels are included because, although they are not actual statements, they are integral parts of your programs.
2. Most described statements begin at the top of a page.

Arithmetic IF Statement

See "IF Statements" on page 127.

ASSIGN Statement

The ASSIGN statement assigns a number (a **statement label**) to an integer variable. See "Statement Labels" on page 207.

Syntax

ASSIGN *stl* **TO** *i*

stl

is the label of an executable statement or a FORMAT statement in the same program unit as the ASSIGN statement.

i

is the name of an integer variable (not an array element) of length 4 that is assigned the statement label *stl*.

The statement label must be the number of a statement that appears in the same program unit as the ASSIGN statement. The statement label must be the number of an executable statement or a FORMAT statement.

Execution of ASSIGN is the only way that a variable can be defined with a statement label. A variable must have been defined with a statement label when it is referred to in an assigned GO TO statement or as a format identifier in an input or output statement. An integer variable defined with a statement label may be redefined with the same or a different statement label or an integer value.

If *stl* is the statement label of an executable statement, *i* can be used in an assigned GOTO statement. If *stl* is the statement label of a FORMAT statement, *i* can be used as the format identifier in a READ, WRITE, or PRINT statement with FORMAT control.

The value of *i* is not the integer constant represented by *stl* and cannot be used as such. To use *i* as an integer, it must be assigned an integer value by an assignment or input statement. This assignment can be done directly or through EQUIVALENCE, COMMON, or argument passing.

Valid ASSIGN Statements:

This program fragment illustrates the use of the ASSIGN statement to assign the statement labels of both an executable statement and a FORMAT statement to variables.

Program fragment:	**Function:**
`10 FORMAT (1X, I4)`	
`   ASSIGN 30 TO LABEL`	Assign statement label 30 to integer variable LABEL.
`   ASSIGN 10 TO IFMT`	Assign FORMAT statement label 10 to integer variable IFMT.
`   NUM = 50`	

Program fragment:	**Function:**
`   GOTO LABEL`	Transfer to statement labeled 30.
`20 WRITE(5, IFMT) NUM`	Write using the FORMAT statement at statement 10.
`30 PRINT *, NUM`	
`   END`	

Invalid ASSIGN Statements:

This program fragment illustrates an invalid use of the ASSIGN statement. The variable set by an ASSIGN statement does not have the integer value representation of the statement label.

The statement `IF (NUM .EQ. LABEL) GOTO 20` is invalid. The results are unpredictable.

```
    ASSIGN 10 TO LABEL
 10 NUM = 10
    IF (NUM .EQ. LABEL) GOTO 20
    NUM = 20
 20 CONTINUE
    END
```

Assigned GO TO Statement

See "GO TO Statements" on page 125.

Assignment Statements

This statement closely resembles a conventional algebraic equation; however, the equal sign specifies a replacement operation rather than equality. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable, array element, character substring, or character variable to the left of the equal sign.

Syntax

a = *b*

a

is a variable, array element, character substring, or character variable.

b

is an arithmetic, character, or logical expression.

An assignment statement is used to obtain the results of calculations. The result of evaluating the expression replaces the current value of a designated variable, array element, character substring, or character variable. There are three types of assignment statements: arithmetic, character, and logical.

Arithmetic Assignment Statement

If *b* is an arithmetic expression, *a* must be an integer, real, or complex variable or an array element.

Figure 21 on page 53 shows the rules for conversion in arithmetic assignment statements, *a*=*b*, where the type of *b* is integer, real or complex.

The correspondence between type declarations and data item lengths in bytes is described in Figure 22 on page 94.

Type of b Type of a	INTEGER*2 INTEGER*4 INTEGER	REAL*4 REAL	REAL*8 Double Precision	REAL*16	COMPLEX*8 COMPLEX	COMPLEX*16	COMPLEX*32
INTEGER*2 INTEGER*4 INTEGER	Assign	Fix and assign	Fix and assign	Fix and assign	Fix and assign real part; imaginary part not used	Fix and assign real part; imaginary part not used	Fix and assign real part; imaginary part not used
REAL*4 REAL	Float and assign	Assign	Real assign	Real assign	Assign real part; imaginary part not used	Real assign real part; imaginary part not used	Real assign real part; imaginary part not used
REAL*8 Double Precision	DP float and assign	DP extend and assign	Assign	DP assign	DP extend and assign real part; imaginary part not used	Assign real part; imaginary part not used	DP assign real part; imaginary part not used
REAL*16	QP float and assign	QP extend and assign	QP extend and assign	Assign	QP extend and assign real part; imaginary part not used	QP extend and assign real part; imaginary part not used	Assign real part; imaginary part not used
COMPLEX*8 COMPLEX	Float and assign to real part; imaginary part set to zero	Assign to real part; imaginary part set to zero	Real assign real part; imaginary part set to zero	Real assign real part; imaginary part set to zero	Assign	Real assign real and imaginary parts	Real assign real and imaginary parts
COMPLEX*16	DP Float and assign to real part; imaginary part set to zero	DP extend and assign to real part; imaginary part set to zero	Assign to real part; imaginary part set to zero	DP assign real part; imaginary part set to zero	DP extend and assign real and imaginary parts	Assign	DP assign real and imaginary parts
COMPLEX*32	QP float and assign to real part; imaginary part set to zero	QP extend and assign to real part; imaginary part set to zero	QP extend and assign real part; imaginary part set to zero	Assign real part; imaginary part set to zero	QP extend and assign real and imaginary parts	QP extend and assign real and imaginary parts	Assign

Figure 21. Conversion Rules for the Arithmetic Assignment Statement a=b.

Terms in Figure 21 are defined as follows:

Assign Transmit the expression value without change. If the expression value contains more significant digits than the variable *a* can hold, the value assigned to *a* is unpredictable.

Real assign Transmit to *a* as much precision of the most significant part of the expression value as REAL*4 data can contain.

DP assign Transmit as much precision of the most significant part of the expression value as double precision (REAL*8) data can contain.

Fix Truncate the fractional portion of the expression value and transform the result to an integer of 4 bytes in length. If the expression value contains more significant digits than an integer 4 bytes in length can hold, the value assigned to the integer variable is unpredictable.

Float Transform the integer expression value to a REAL*4 number, retaining in the process as much precision of the value as a REAL*4 number can contain.

DP float Transform the integer expression value to a double precision (REAL*8) number.

DP extend Extend the real value to a double precision (REAL*8) number.

QP float Transform the integer expression value to a REAL*16 number

QP extend Extend the real value to a REAL*16 number.

Character Assignment Statement

If *b* is a character expression, *a* must be a character variable, character array element, or character substring.

None of the character positions being defined in *a* must be referenced in *b* directly or through associations of variables (that is, using COMMON, EQUIVALENCE, or argument passing).

The lengths of *a* and *b* may be different. The characters of *b* are moved from left to right into the corresponding character positions of *a*. If *a* has more positions than there are characters in *b*, the rightmost positions of *a* are filled with blanks. If *a* has fewer positions than there are characters in *b*, only the leftmost characters of *b* are moved to fill the positions of *a*.

A character variable, character array element, or character substring (*a*) may also be assigned a value by a WRITE statement to an internal file with unit=*a*.

If you are doing assignment operations on double-byte characters, you should use the ASSIGNM service subroutine. See "ASSIGNM Subroutine" on page 277 for more details.

Logical Assignment Statement

If *b* is a logical expression, *a* must be a logical variable or a logical array element. The value of *b* must be either true or false.

Assignment Statement Examples

Assume the type of the following data items has been specified:

Variable Name	Type	Length
I, J, K	Integer	4, 4, 2
A, B, C, D	Real	4, 4, 8, 8
E	Complex	8
F(1),...F(5)	Real array elements	4
G, H	Logical	4, 4
CHAR1	Character	10

The following examples illustrate valid assignment statements using constants, variables, and array elements as defined above.

Statement	Description
A = B	The value of A is replaced by the current value of B.
K = B	The value of B is converted to an integer value, and the value of K is truncated on the left to two bytes.
A = I	The value of I is converted to a real value, and replaces the value of A.
I = I + 1	The value of I is replaced by the value of I + 1.
E = I**J+D	I is raised to the power J and the result is converted to a real value to which the value of D is added. This result replaces the real part of the complex variable E. The imaginary part of the complex variable is set to 0.
A = C*D	The most significant part of the product of C and D replaces the value of A.
A = E	The real part of the complex variable E replaces the value of A.
E = A	The value of A replaces the value of the real part of the complex variable E; the imaginary part is set equal to 0.
G = .TRUE.	The value of G is replaced by the logical value true.
H = .NOT.G	If G is true, the value of H is replaced by the logical value false. If G is false, the value of H is replaced by the logical value true.
G = 3..GT.I	The value of I is converted to a real value; if the real constant 3.0 is greater than this result, the logical value true replaces the value of G. If 3.0 is not greater than the converted I, the logical value false replaces the value of G.
E = (1.0,2.0)	The value of the complex variable E is replaced by the value of the complex constant (1.0,2.0). The statement E = (A,B), where A and B are real variables, is invalid. The mathematical function subprogram CMPLX can be used to solve this problem. See Chapter 5, "Intrinsic Functions" on page 243
F(1) = A	The value of element 1 of array F is replaced by the value of A.
E = F(5)	The real part of the complex constant E is replaced by the value of array element F(5). The imaginary part is set equal to 0.
C = 99999999.0	Even though C is of length 8, the constant having no exponent is considered to be of length 4. Thus the number will not have the accuracy that may be intended. If the basic real constant were entered as 99999999.0D0, the converted value placed in the variable C would be a closer approximation to the entered basic real decimal constant, because 15 decimal digits can be represented in 8 bytes.
CHAR1= 'ABCDEFGHIJ'	CHAR1 contains the value 'ABCDEFGHIJ' because CHAR1 is of length 10, and the constant is of length 10.
CHAR1='ABC'	CHAR1 contains the value 'ABCƀƀƀƀƀƀƀ' because CHAR1 is of length 10, and the constant is only of length 3; thus CHAR1 is padded with blanks.

Statement	Description
CHAR1= 'ABCDEFGHIJKL'	CHAR1 contains the value 'ABCDEFGHIJ' because CHAR1 is of length 10, and the constant is of length 12; the last two characters in the constant are not moved into CHAR1.
CHAR1= 'FGHIJ'//'ABCDE'	CHAR1 contains the value 'FGHIJABCDE', the result of the concatenation operation.

AT Statement

The AT statement identifies the beginning of a debug packet and indicates the point in the program at which debugging statements are to be inserted.

Syntax

AT *stl*

stl

is the statement label of an executable statement in the program unit or function or subroutine subprogram to be debugged.

The debugging operations specified within the debug packet are performed prior to the execution of the statement indicated by the statement label (*stl*) in the AT statement.

The statement label cannot be specified in another debug packet.

The AT statement identifies the beginning of a debug packet and the end of the preceding packet (if any) unless this is the last packet, in which case it is ended by the END DEBUG statement. There may be many debug packets for one program or subprogram.

For more on debug packets and for examples of the AT statement, see "DEBUG Statement" on page 71.

BACKSPACE Statement

The BACKSPACE statement, when first issued, positions a sequentially accessed file to the beginning of the FORTRAN record last written or read. A subsequent BACKSPACE statement will reposition the file to the beginning of the preceding record.

The BACKSPACE statement reestablishes the position of a keyed file to a point prior to the current file position. Following the BACKSPACE statement, you can use a sequential retrieval statement to read the record to which the file was positioned.

Syntax

BACKSPACE *un*

BACKSPACE
    ([UNIT=]*un*
    [,IOSTAT=*ios***]**
    [,ERR=*stl***])**

UNIT=*un*

un is the external unit identifier. It is an integer expression of length 4, whose value must be zero or positive. *un* is required.

If the second form of the statement is used, *un* can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is specified, all the specifiers can appear in any order.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error is detected. VSAM return and reason codes are placed in *ios*. IOSTAT=*ios* is optional.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the BACKSPACE statement. If an error is detected, control is transferred to *stl*.

Valid BACKSPACE Statements:

```
BACKSPACE 08

BACKSPACE (05,ERR=0300)

BACKSPACE (UNIT=12,IOSTAT=errst,ERR=300)

BACKSPACE (ERR=300,UNIT=12)

BACKSPACE(UNIT=2*IN+2)

BACKSPACE(IOSTAT=IOS,ERR=99999,UNIT=2*IN-10)
```

Invalid BACKSPACE Statements:

```
BACKSPACE 08,ERR=235      Parentheses must be specified.

BACKSPACE (ERR=235,08)    UNIT= must be specified when un is not first.
```

When the BACKSPACE statement is encountered, the unit specified by *un* must be connected to an external file for sequential or keyed access. (See *VS FORTRAN Version 2 Programming Guide*.) If the unit is not connected, an error is detected.

But when the NOOCSTATUS run-time option is in effect, the unit does not have to be connected to an external file for sequential access. For more information on the NOOCSTATUS option, see *VS FORTRAN Version 2 Programming Guide*.

A BACKSPACE statement positions an external file connected for sequential access to the beginning of the preceding record. If there is no preceding record, the BACKSPACE statement has no effect. The BACKSPACE statement must not be used with external files using list-directed formatting.

An external file connected for sequential access can be extended if the execution of an ENDFILE statement or the detection of an end-of-file is immediately followed by the execution of a BACKSPACE and a WRITE statement on this file.

If the external file connected for sequential access is at the end-of-file, either after an ENDFILE operation or after a READ that resulted in end-of-file, two BACKSPACE statements are necessary to position the data set to the beginning of its last logical record.

A BACKSPACE issued to a file connected for keyed access positions the file to the beginning of the first record whose key value is the same as that in the record that precedes the current file position. If there is no preceding record, the file position remains at the beginning of the file.

The BACKSPACE statement must not be used with external files written using NAMELIST. If it is used, the result is unpredictable.

The BACKSPACE statement may be used with asynchronous READ and WRITE statements, provided that any input or output operation on the file has been completed by the execution of a WAIT statement. A WAIT statement is not required to complete the BACKSPACE operation.

If an error is detected, transfer is made to the statement label designated by the ERR specifier. If IOSTAT=*ios* is specified, a positive integer value is assigned to *ios* when an error is detected. Execution continues with the statement label designated by the ERR specifier (if present) or with the next statement if no ERR specifier is included on the BACKSPACE statement.

BLOCK DATA Statement

The BLOCK DATA subprogram initializes values for variables and array elements in named common blocks.

Syntax

BLOCK DATA [*name*]

name

is the name of the block data subprogram. This name is optional. It must not be the same as the name of another subprogram, a main program, or a common block name in the executable program. There can be only one unnamed block data subprogram in an executable program.

To initialize variables in a named common block, a separate subprogram must be written. This separate subprogram contains only the BLOCK DATA, IMPLICIT, PARAMETER, DATA, COMMON, DIMENSION, SAVE, EQUIVALENCE, and END statements, comment lines, and explicit type specification statements associated with the data being defined. This subprogram is not called; its presence provides initial data values for named common blocks. Data may not be initialized in unnamed common blocks.

The BLOCK DATA statement must appear only as the first statement in the subprogram. Statements that provide initial values for data items cannot precede the COMMON statements that define those data items.

Any main program or subprogram using a named common block must contain a COMMON statement defining that block. If initial values are to be assigned, a block data subprogram is necessary.

A particular common block may not be initialized in more than one block data subprogram.

Local variables cannot be declared in a BLOCK DATA statement. A variable (or array) equivalenced to another in a common block is considered to be in that common block.

All elements of a named common block must be listed in the COMMON statement, even though they are not all initialized. For example, the variable A in the COMMON statement in the following block data subprogram does not appear in the DATA statement.

Example 1:

```
BLOCK DATA
COMMON /ELN/C,A,B
COMPLEX C
DATA C/(2.4,3.769)/,B/1.2/
END
```

Data may be entered into more than one common block in a single block data subprogram.

Example 2:

```
BLOCK DATA VALUE1
COMMON /ELN/ C,A,B
COMMON /RMG/ Z,Y
COMPLEX C
DOUBLE PRECISION Z
DATA C /(2.4, 3.769)/
DATA B /1.2/
DATA Z /7.64980825D0/
END
```

As a result of the operation in this example, in BLOCK DATA named VALUE1,

```
COMMON/ELN/C,A,B
```

will have the complex variable C real part initialized to 2.4 and the imaginary part initialized to 3.769. The variable A will not be initialized and B will be initialized to 1.2.

```
COMMON/RMG/Z,Y
```

will have the double precision variable Z initialized with the double precision constant 7.64980825 and Y will not be initialized.

Block IF Statement

See "IF Statements" on page 127.

CALL Statement

The CALL statement:

1. Evaluates actual arguments on the CALL statement that are expressions
2. Passes actual arguments that will be associated with dummy arguments defined in the subroutine subprogram
3. Transfers control to a subroutine subprogram

Syntax

CALL *name* **[([***arg1* **[,** *arg2* **]...])]**

name
is the name of a subroutine subprogram or an entry point. This name may be a dummy argument in a SUBROUTINE statement or in an ENTRY statement.

arg
is an actual argument that is being supplied to the subroutine subprogram. The argument may be a variable, array element, or array name; a constant; an arithmetic, logical, or character expression; a function or subroutine name; or an asterisk (*) followed by the statement label of an executable statement that appears in the same program unit as the CALL statement.

If no actual argument is specified, the parentheses may be omitted.

The CALL statement transfers control to a subroutine subprogram and passes actual arguments that will be associated with dummy variables. This association is done by passing the addresses of the actual arguments to the subroutine subprogram.

The CALL statement can be used in a main program, a function subprogram, or a subroutine subprogram, but a subprogram must not refer to itself directly or indirectly and must not refer to the main program. A main program cannot call itself.

If *name* is a dummy argument in a subprogram containing CALL *name*, this CALL statement can be executed only if the subprogram is given control at one of its entry points where *name* appears in the list of dummy arguments. (See "EXTERNAL Statement" on page 95.)

Valid CALL statements:

Example 1:

Assume that the following subroutine definition has been made:

```
SUBROUTINE SUB1
⋮
END
```

The next two statements are valid ways to call a subroutine with no arguments.

```
CALL SUB1
CALL SUB1()
```

Example 2:

Assume that this subroutine definition has been made:

```
SUBROUTINE SUB2(A, B, C)
REAL A
REAL B(*)
REAL C(2, 5)
⋮
END
```

And that these variables have been declared:

```
DIMENSION W(10), X(10), Z(5)
REAL Y
```

Example with a variable and two array names:

```
CALL SUB2(Y, W, X)
```

Example with an array element and two array names:

```
CALL SUB2(Z(3), X, W)
```

Example with a constant and two array names:

```
CALL SUB2(2.5, W, X)
```

Example with an expression and two array names:

```
CALL SUB2(5*Y, X, W)
```

Note that the size of an actual array passed as an argument must be larger than or equal to the size of the corresponding dummy array. For information on array layouts, see "Subscripts" on page 25.

Example 3:

For the following examples, assume this subroutine definition has been defined:

```
SUBROUTINE SUB3(LOGL)
LOGICAL LOGL
⋮
END
```

With this variable declaration:

```
LOGICAL L
```

Example using a logical variable:

```
CALL SUB3(L)
```

Example using a logical constant:

```
CALL SUB3(.FALSE.)
```

Example using a logical expression:

```
CALL SUB3(X(5) .EQ. Y)
```

Example 4:

Assume the following subroutine definition was made:

```
SUBROUTINE SUB4(CHAR)
CHARACTER*(*) CHAR
⋮
END
```

With the following declaration:

```
CHARACTER*5 C1, C2
```

Example using a character variable:

```
CALL SUB4(C1)
```

Example using a character expression:

```
CALL SUB4(C1 // C2)
```

Example 5:

Assume subroutines SUB5 and SUB6 are as follows:

```
SUBROUTINE SUB5(SUBX, X, Y, FUNCX)
EXTERNAL SUBX, FUNCX
Z = FUNCX(X, Y)
CALL SUB6(SUBX)
:
END

SUBROUTINE SUB6(SUBY)
EXTERNAL SUBY
:
CALL SUBY
END
```

With the following declaration:

```
EXTERNAL SUBZ, FUNCA
```

Example of passing a subroutine name and a function name:

```
CALL SUB5(SUBZ, 1.0, 2.0, FUNCA)
```

Example 6:

```
SUBROUTINE SUB7 (A, B, *, *, *)
:
IF(A .LT. 0.0) RETURN 1
IF(A .EQ. 0.0) RETURN 2
RETURN 3
END
```

Example of passing statement labels. Execution will continue at the statement labeled 100, 200, or 300 depending on the value of the RETURN specifier. Otherwise, execution will continue at the statement after the call.

```
CALL SUB7(X(3), LOG(Z(2)), *100, *200, *300)
```

Invalid CALL statements: Assume the same subroutine definitions as Example 5 above. The following example results indirectly in a call by one subroutine to itself. This is invalid, but cannot be checked by the compiler.

```
CALL SUB5(SUB6, X(5), Y, COS)
```

CHARACTER Type Statement

See "Explicit Type Statement" on page 91.

CLOSE Statement

A CLOSE statement disconnects a unit.

```
Syntax

CLOSE
      ( [UNIT=]un
      [, ERR=stl ]
      [, STATUS=sta ]
      [, IOSTAT=ios ] )
```

UNIT=*un*

un is the external unit identifier. It is an integer expression of length 4, whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the CLOSE statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the CLOSE statement. If an error is detected, control is transferred to *stl*.

STATUS=*sta*

sta is a character expression whose value (when any trailing blanks are removed) must be KEEP or DELETE. *sta* determines the disposition of the file that is connected to the specified unit.

If the STATUS specifier is omitted, the assumed value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case the assumed value is DELETE. For a discussion on the concept of file status, see page 151.

Note: The run-time options OCSTATUS and NOOCSTATUS affect the operation of the CLOSE statement. For details on these options, see *VS FORTRAN Version 2 Programming Guide.*

If KEEP is specified for a file that exists, the file continues to exist after the execution of the CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of the CLOSE statement. If DELETE is specified, VS FORTRAN attempts to delete the file.

If KEEP is specified for a file whose status prior to execution of the CLOSE statement is SCRATCH, an error is detected.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*.

Execution of a CLOSE statement that refers to a unit may occur in any program unit of an executable program and need not occur in the same program unit as the execution of an OPEN statement referring to that unit. When the CLOSE statement is encountered, the unit specified by *un* need not be connected to a file. If the unit is connected, the file need not exist.

After a **unit** has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program to the same file or a different file.

After a named file has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program to the same unit or a different unit; an unnamed file may only be reconnected to the same unit. (See "OPEN Statement" on page 151.)

When execution ends normally, all units are disconnected. Each unit is closed with the status KEEP, unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with the status DELETE.

Assume that the type of the following variables has been specified as follows:

Variable Names	**Type**	**Length**
IN, IACT, Z	INTEGER	4
DELETE, STATUS	CHARACTER·	6

and that

```
DELETE = 'DELETE'
```

The following statements are valid:

Example 1:

```
CLOSE(6+IN)

CLOSE(Z*IN+2)

CLOSE(Z*IN+3,STATUS=DELETE)

CLOSE(IOSTAT=IACT,ERR=99999,STATUS='KE'//'EP ',UNIT=0)
```

Example 2:

```
STATUS='KEEP'

CLOSE(UNIT=9,STATUS=DELETE)

CLOSE(UNIT=10,STATUS=STATUS)

CLOSE(UNIT=11,STATUS='KEEP')
```

Comments

Comments provide documentation for a program. If you are working with fixed format source code, you must use fixed-form comments; if you are working with free format source code, you must use free-form comments.

For more information on how to use comments in your program, see "Comments" on page 12.

COMMON Statement

The COMMON statement makes it possible for two or more program units to share storage and to specify the names of variables and arrays that are to occupy the area.

Syntax

COMMON [/**[***name1***]**/ **]** *list1* **[[,]** /**[***name2***]**/ *list2* **...]**

name

is an optional common block name. These names must always be enclosed in slashes. They cannot be the same as names used in PROGRAM, SUBROUTINE, FUNCTION, ENTRY, or BLOCK DATA statements. They cannot be intrinsic function names that are referenced in the same program unit.

The form // (with no characters except, possibly, blanks between the slashes) denotes blank common. If *name*1 denotes blank common, the first two slashes are optional.

The comma preceding the common block name designator /*name*/ is optional.

list

is a list of variable names or array names that are not dummy arguments. If a variable name is also a function name, subroutine name, or entry name, it must not appear in the list. If the list contains an array name, dimensions may also be declared for that array. (See "DIMENSION Statement" on page 76.)

A given common block name may appear more than once in a COMMON statement, or in more than one COMMON statement in a program unit.

Blank and named common entries appearing in COMMON statements are cumulative throughout the program unit. Consider the following two COMMON statements:

```
COMMON A, B, C /R/ D, E /S/ F

COMMON G, H /S/ I, J /R/R//W
```

These two statements have the same effect as the single statement:

```
COMMON A, B, C, G, H, W /R/ D, E, R /S/ F, I, J
```

Character and noncharacter data types can be mixed in a common block.

Although the entries in a COMMON statement can contain dimension information, object-time dimensions may never be used. A common block resides in a fixed location in storage during the execution of a program. The length of a blank common can be extended by using an EQUIVALENCE statement, but only by adding beyond the last entry. Arrays may be declared in a common statement, but they must be actual arrays.

In the following example, the complex variable, CV, and the real array, RV, refer to the same storage locations.

The statement: RV(2) = 1.2 will assign the value of 1.2 to the imaginary part of CV.

```
Main Program          Subroutine

COMMON CV             SUBROUTINE SUB
COMPLEX*8 CV          COMMON RV(2)
  .                     .
  .                     .
  .                     .
CALL SUB              RV(2) = 1.2
  .                     .
  .                     .
  .                     .
STOP                  RETURN
END                   END
```

Blank and Named Common Blocks

Variables and arrays may be placed in separate common blocks by giving them distinct common block names. Those blocks that have the same name occupy the same storage area. The name cannot be the same as the main program name, subprogram name, or entry name.

The variables and arrays of a common block may be mixed character and non-character data types.

Naming these separate blocks permits a calling program to share one common block with one subprogram and another common block with another subprogram. It also makes it easier to document the program.

The differences between *blank* and *named* common blocks are:

- There is only one *blank* common block in an executable program, and it has no name.

  There may be many *named* common blocks, each with its own name.

- *Blank* common blocks may have different lengths in different program units.

  Each program unit that uses a *named* common block must define it to be of the same length.

- Variables and array elements in a *blank* common block cannot be assigned initial values.

- Variables and array elements in a *named* common block may be assigned initial values by DATA statements or by explicit type specification statements in a block data subprogram.

Variables that are to be placed in a *named* common block are preceded by the common block name enclosed in slashes. For example, the variables A, B, and C are placed in the named common block, HOLD, by the following statement:

```
COMMON /HOLD/ A,B,C
```

In a COMMON statement, a *blank* common block is distinguished from a *named* common block by placing two consecutive slashes before the variables (or, if the variables appear at the beginning of the COMMON statement, by omitting any common block name). For example,

```
COMMON A, B, C /ITEMS/ X, Y, Z / / D, E, F
```

The variables A, B, C, D, E, and F are placed in a *blank* common block in that order; the variables X, Y, and Z are placed in the named common block, ITEMS.

COMPLEX Type Statement

See "Explicit Type Statement" on page 91.

Computed GO TO Statement

See "GO TO Statements" on page 125.

CONTINUE Statement

The CONTINUE statement is an executable control statement that takes no action. It can be used to designate the end of a DO loop, or to label a position in a program.

Syntax

CONTINUE

CONTINUE is a statement that may be placed anywhere in the source program (where an executable statement may appear) without affecting the sequence of execution. It may be used as the last statement in the range of a DO loop. Using it to end a DO loop will allow you to avoid using a statement that is prohibited from ending a DO loop.

DATA Statement

The DATA statement defines initial values of variables, array elements, arrays, and substrings.

Syntax

DATA *list1* /*clist1*/ **[[,]** *list2* /*clist2*/ **...]**

list

is a list of variables, array elements, arrays or substrings, and implied DO lists. (See "Implied DO in a DATA Statement" on page 80.) The comma preceding *list2...* is optional.

Subscript and substring expressions used in each *list* can contain only integer constants or names of integer constants.

clist

is a list of constants or the names of constants. Integer and real constants may, optionally, be signed. Any of these constants may be preceded by *r**, where *r* is a nonzero unsigned integer constant or the name of such a constant. When the form *r** appears before a constant, it indicates that the constant is to be repeated *r* times.

A DATA initialization statement is not executable. DATA statements must follow all other specification statements which refer to the same item; however, DATA statements may be intermixed with all specification statements except for the IMPLICIT statement.

There must be a one-to-one correspondence between the total number of elements specified or implied by the list *list* and the total number of constants specified by the corresponding list *clist* after application of any replication factors, *r*.

Integer, real, and complex variables or array elements must be initialized with integer, real, or complex constants; conversions take place according to the arithmetic assignment rules, if necessary.

A hexadecimal constant can be used to initialize any type of variable or array element. When initializing a variable of character type with hexadecimal data, the data is right-justified and extended to the left with zeros to fill the substring being initialized.

If a hexadecimal constant initializes a complex data type, one constant is used that initializes both the real and the imaginary parts, and the constant is not enclosed in parentheses. If the constant is smaller than the length (in bytes) of the entire complex entity, zeros are added on the left. If the constant is larger, the leftmost hexadecimal digits are truncated.

A Hollerith constant can be used to initialize a noncharacter variable or array element.

A logical variable or logical array can be initialized with T instead of .TRUE. and F instead of .FALSE..

Character items can be initialized by character data. Each character constant initializes exactly one variable, one array element, or one substring. If a character constant contains more characters than the item it initializes, the additional rightmost characters in the constant are ignored. If a character constant contains fewer characters than the item it initializes, the additional rightmost characters in the item are initialized with blank characters. (Each character represents one byte of storage.)

A variable in a blank common cannot be defined with an initial value. A variable in a named common block can be initially assigned a value only in a block data subprogram. Because of this constraint, entities that are associated with each other through COMMON or EQUIVALENCE statements are considered to be the same entity.

Valid DATA Statements:

Example 1:

```
LOGICAL L(4)
CHARACTER*4 C
DIMENSION D(50),F(5),G(9)
DATA A, B, S/5.0,6.1,7.3/,D/25*1.0,25*2.0/,E/5.1/
DATA F/5*1.0/, G/9*2.0/, L/2*.TRUE.,2*F/, C/'FOUR'/
```

Example 2:

```
CHARACTER*4 CC(5)
DATA CC(1)(1:2)/'AB'/,CC(1)(3:4)/'CD'/
DATA CC(2)/ZC5C6C7C8/,I/ZF8/,R/Z00/
```

Example 3:

```
PARAMETER (DEGI=10.2,NRANGE=7)
DATA DEG/DEGI/,IRANGE/NRANGE/
```

Example 4:

```
DIMENSION A(5)
DATA A(1),A(2),A(3),A(4),A(5)/1.0,2.0,3.0,4.0,5.0/
```

Example 5:

```
DIMENSION ARRAYE(10,10)
DATA ((ARRAYE(I,J),I=1,10),J=1,10)/100*0.0/
```

DEBUG Statement

The DEBUG statement sets the conditions for operation of the debug facility and designates debugging operations that apply to the entire program unit (such as subscript checking).

Syntax

DEBUG *option1* **[,** *option2...***]**

An *option* may be any of the following:

UNIT (*un***)**
un is an integer constant that represents a unit number. All debugging output is placed in this file, which is called the debug output file. If this option is not specified, any debugging output is placed in the installation-defined output file. All unit definitions within an executable program must refer to the same unit.

SUBCHK (*a1*, *a2*, **...)**
a is an array name. The validity of the subscripts used with the named arrays is checked by comparing the subscript combination with the size of the array. If the subscript value exceeds the size of the array, a message is placed in the debug file. Program execution continues, using the incorrect subscript. If the list of array names is omitted, all arrays in the program are checked for valid subscript usage. If the entire option is omitted, no arrays are checked for valid subscripts.

TRACE
This option must be in the DEBUG specification statement of each program or subprogram for which tracing is desired. If this option is omitted, there can be no display of program flow by statement label within this program. Even when this option is used, a TRACE ON statement must appear in the first debug packet in which tracing is desired.

INIT (*i1*, *i2*, **...)**
i is the name of a variable or an array that is to be displayed in the debug output file only when the variable or the array elements are assigned a value. If *i* is a variable name, the name and value are displayed whenever the variable is assigned a new value in either an assignment, a READ, or an ASSIGN statement. If *i* is an array name, the array element is displayed. If the list of names is omitted, a display occurs whenever the value of a variable or an array element is assigned a value. If the entire option is omitted, no display occurs when values are assigned.

SUBTRACE
This option specifies that the name of this subprogram is to be displayed whenever it is entered. The return message is to be displayed whenever execution of the subprogram is completed.

The options in a DEBUG statement may be given in any order and must be separated by commas.

All debugging statements must precede the first statement of the program unit to which they refer.

In a subroutine, the debug statements must appear immediately before the SUBROUTINE statement. In a function subprogram, the debug statements must appear immediately before the FUNCTION statement. The required statement sequence is:

1. DEBUG statement
2. Debug packets
3. END DEBUG statement
4. First of the source program statements of a program unit to be debugged

A debug packet begins with an AT statement and ends when either another AT statement or an END DEBUG statement is encountered.

Debug statements are written in either fixed form or free form and follow the same rules as other FORTRAN statements.

In addition to the language statements, the following debug statements are allowed:

TRACE ON
TRACE OFF
DISPLAY

All FORTRAN statements are allowed in a debug packet, except as listed in "Considerations when Using DEBUG," below.

Considerations when Using DEBUG

When setting up a debug packet, use the following precautions:

- Any DO loops or block IF, ELSE IF, or ELSE statements initiated within a debug packet must be wholly contained within that packet.
- Statement labels within a debug packet must be unique. They must be different from statement labels within other debug packets and within the program being debugged.
- An error in a program should not be corrected with a debug packet; when the debug packet is removed, the error remains in the program.
- No specification statements can appear in a debug packet; nor can any of the following statements:

  BLOCK DATA
  ENTRY
  FUNCTION
  PROGRAM
  statement function
  SUBROUTINE

- The program being debugged must not transfer control to any statement label defined in a debug packet; however, control may be returned from a packet to any point in the program being debugged. In addition, no debug packet may refer to a label defined in another debug packet. A debug packet may contain a RETURN, STOP, or CALL statement.
- The SUBCHK function of DEBUG does proper subscript checking of an array if, and only if, that array is a single-dimensioned array with a lower bound of 1. If the lower bound is not 1 and an error is detected, the message will give the index to the element as if it had a lower bound of 1. If multidimen-

sional arrays are being checked for valid subscripts, the array is perceived to be a single-dimensioned array of the appropriate number of array elements. The subscripts are evaluated and the check indicates whether you are referencing an array element within the range of the array, but not whether one of the subscripts is invalid. Individual subscripts are not checked for their valid range.

Thus, if array A is dimensioned as A(5,6) and a reference is made to A(K,2), where K is 7, the SUBCHK function will not flag this because the subscript value yields an element *within* array A. The values of the first and second subscripts are *not* checked for having values of 1 through 5 or 1 through 6, respectively.

DEBUG Examples:

Example 1:

```
   DEBUG UNIT(6),SUBCHK
   END DEBUG
   PROGRAM TEST
     .
     .
     .
   END
```

This example checks all arrays for valid subscripts.

Example 2:

```
   DEBUG UNIT(6)
   AT 11
   WRITE(6,21)A,B,C
21 FORMAT(1X,'A=',I10,'B=',I10,'C=',I10)
   END DEBUG
     .
     .
     .
   INTEGER A,B,C
     .
     .
     .
10 B=A* SQRT(FLOAT(C))
11 IF(B)40,50,60
     .
     .
     .
```

The values of A, B, and C are to be examined as they were at the completion of the arithmetic operation in statement 10. Therefore, the statement label specified in the AT statement is 11. The values of A, B, and C are written to the file connected to unit 6.

Example 3:

```
       DEBUG TRACE, UNIT(6)
       AT 10
       TRACE ON
       AT 25
       TRACE OFF
       AT 35
       DISPLAY C
       TRACE ON
       END DEBUG
         .
         .
         .
10     A=2.0
15     L= 1
20     B = A + 1.5
25     DO 30 I = 1,5
         .
         .
         .
30     CONTINUE
35     C = B + 3.415
40     D=C**2
45     CALL SUB1(D,L,R)
       STOP
       END

       DEBUG SUBTRACE,TRACE
       AT 4
       TRACE ON
       END DEBUG
       SUBROUTINE SUB1(X,I,Y)
         .
         .
         .
4      Y=FUNC1(X-INT(X))
       WRITE (6,*) Y
         .
         .
         .
       RETURN
       END

       DEBUG SUBTRACE,TRACE
       AT 100
       TRACE ON
       END DEBUG
       FUNCTION FUNC1(Z)
         .
         .
         .
100    FUNC1 = COS(Z) + SIN(Z)
         .
         .
         .
       RETURN
       END
```

After statement 10 is encountered, tracing begins, as specified by the TRACE ON statement in the first debug packet. After statement 25 is encountered, tracing stops, as specified by the TRACE OFF statement in the second debug packet. After statement 35 is encountered, tracing begins again and the value of C is written to the debug output file, as specified in the third debug packet.

When SUB1 is entered, the words "SUBTRACE SUB1" appear in the output because of the SUBTRACE option on the DEBUG statement in subroutine SUB1. When statement 4 is encountered, tracing begins. When FUNC1 is entered, the words "SUBTRACE FUNC1" appear in the output. When FUNC1 is exited, the words "SUBTRACE RETURN FROM FUNC1" appear in the output, and, similarly, at exit from SUB1, the words "SUBTRACE RETURN FROM SUB1" appear. Note that the output from the WRITE statement in SUB1 will go to the same unit (6) as the DEBUG output.

DELETE Statement

The DELETE statement removes a record from a file connected for keyed access. It removes the record retrieved by an immediately preceding READ operation. No other operation, such as BACKSPACE or WRITE, can be issued for the same file between the READ and DELETE statements.

Syntax

DELETE *un*

DELETE
    ([UNIT=]*un*
    [, IOSTAT=*ios* **]**
    [, ERR=*stl* **])**

UNIT=*un*

un is the external unit identifier. It is an integer expression of length 4 whose value must be zero or positive. *un* is required.

If the second form of the statement is used, *un* can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear immediately following the left parenthesis. The other specifiers can appear in any order. If UNIT= is included on the DELETE statement, all the specifiers can appear in any order.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. It is set positive if an error is detected; it is set to zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*. IOSTAT=*ios* is optional.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the DELETE statement. If an error is detected, control is transferred to *stl*. ERR=*err* is optional.

DIMENSION Statement

The DIMENSION statement specifies the name and dimensions of an array.

Syntax

DIMENSION *a1*(*dim1*) **[**, *a2*(*dim2*) ... **]**

a

is an array name.

dim

is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array in the form:

```
e1:e2
```

or

```
e2
```

where:

e1

is the lower dimension bound. It is optional. If *e1* (with its following colon) is not specified, its value is assumed to be 1.

e2

is the upper dimension bound and must always be specified.

(For rules governing dimension bounds, see "Size and Type Declaration of an Array" on page 26.)

Each *a* in a DIMENSION statement declares that *a* is an array in that program unit. Array names and their bounds may also be declared in COMMON statements and in type statements. Only one declaration of the array name (*a*) as an array is permitted in a program unit.

Valid DIMENSION Statements:

```
DIMENSION A(10), ARRAY(5,5,5), LIST(10,100)

DIMENSION A(1:10), ARRAY(1:5,1:5,1:5), LIST(1:10,1:100)

DIMENSION B(0:24), C(-4:2), DATA(0:9,-5:4,10)

DIMENSION G(I:J,M:N)

DIMENSION ARRAY (M*N:I*J)

DIMENSION ARRAY (M*N:I*J,*)
```

DISPLAY Statement

The DISPLAY statement displays data in NAMELIST output format. It may appear anywhere within a debug packet.

Syntax

DISPLAY *list*

list

is a list of variable or array names separated by commas.

The DISPLAY statement eliminates the need for FORMAT or NAMELIST and WRITE statements to display the results of a debugging operation. The data is placed in the debug output file.

The effect of a DISPLAY list statement is the same as the following source language statements:

NAMELIST /*name*/ *list*

WRITE (*un*, *name*)

where *name* is the same in both statements.

Array elements, dummy arguments, and substring references may not appear in the list.

For examples and explanations of the DISPLAY statement, see "DEBUG Statement" on page 71.

DO Statement

The DO statement controls the execution of the statements that follow it, up to and including the statement that denotes the end of the DO loop. These statements are called the "range of the DO" or a "DO loop."

Syntax

DO [*stl* **[,]]** *i*=*e1*, *e2* **[**,*e3***]**

stl

is the statement label of an executable statement, in the same program unit as the DO statement, that denotes the end of the DO loop. If you do not specify *stl*, you must use an END DO statement to indicate the end of the DO loop. If you code both an END DO statement and *stl*, then *stl* must be the statement label of the END DO statement. The statement at *stl* must not be one of the following:

- An unconditional or assigned GOTO
- block IF, ELSE, ELSE IF, END IF, arithmetic IF
- INCLUDE
- STOP, RETURN, END
- Another DO statement

i
: is an integer, real, or double precision variable (not an array element) called the DO variable.

e1, *e2*, and *e3*
: are integer, real, or double precision arithmetic expressions that define the DO-loop iteration. *e1* is the initial value; *e2* is the test value. The test value, also called the terminal value, is the value compared to the current value for the DO variable (the value for which the DO loop processing is to end). *e3*, the increment, is optional and cannot have a value of 0; if it is omitted, its value is assumed to be 1, and the preceding comma must be omitted. The control parameters, *m1*, *m2*, and *m3*, are established by evaluating *e1*, *e2*, and *e3*, respectively. Evaluation includes, if necessary, conversion to the type of the DO variable according to the rules for arithmetic conversion.

  Note: The use of real numbers for DO statement parameters (initial value, test value, or increment) can lead to unexpected results. A floating point number is only an approximation of the real number it represents. If a DO statement parameter cannot be represented exactly in a computer, the iterations through a loop may not occur as expected, and the parameter's use is not recommended.

The DO statement is processed once. When the DO statement is processed, *i* is initialized to the value of *m1*.

The statements in the range of the DO are executed only if:

m1 is less than or equal to *m2*, and *m3* is greater than 0

or

m1 is greater than or equal to *m2*, and *m3* is less than 0.

If one of the above relationships is true, the statements in the range of the DO are executed, using the initial value of *i* (initialized from *m1*); on each succeeding iteration, *i* is incremented by the value of *m3*. The number of iterations that can be executed, called the iteration count, is the value of:

MAX (INT((*m2* - *m1* + *m3*) / *m3*), 0).

When the iteration count is 0, execution continues with the statement following the last statement of the range of the DO, or the next outer DO if the statement labeled *stl* is shared by more than one DO.

If one of the above relationships is *not* true, execution continues with the statement following the last statement of the range of the DO, or the next outer DO if the statement labeled *stl* is shared by more than one DO.

The DO variable should not be redefined within the range of the DO loop. No transfers may be made to any of the executable statements within the range of the DO by statements outside the range of the DO.

You can nest DO loops; if you nest one DO loop within another, you must include the range of the inner DO loop entirely within the range of the outer DO loop. You can use the same terminal statement for both the inner and the outer DO statement ranges, but remember that this shared terminal statement is actually part of the inner loop.

You can also code a DO WHILE loop within a DO loop and a DO loop within a DO WHILE loop. You must include the range of the inner loop entirely within the range of the outer loop. The loops may not use the same terminal statement.

If you code a DO loop within a block IF, ELSE IF, or ELSE block, make sure that the range of the DO loop is completely contained within that block. If you code an IF-THEN-ELSE structure within a DO loop, ensure that the entire structure, including END IF, is within the range of the DO loop. (You cannot use the END IF as the terminal statement for the DO loop.)

When you execute a DO statement, the DO loop becomes active. It remains active until one of the following occurs:

- The loop processes completely.
- The program processes a RETURN statement within the DO loop's range.
- A transfer is made out of the DO loop's range.
- Any STOP statement is processed anywhere in the loop.
- Program processing is ended because of an error condition.

Valid DO Statements:

The following program fragment illustrates the use of a negative increment.

```
      DIMENSION IA(20)
      IEND = 20
      INCR = 1
      DO 10, I = IEND/2, 1, -INCR
   10 IA(I) = IA(I) + IA(I+1)
```

The iteration count for the above example is 10; that is,

Iteration Count = MAX(INT((1 - 10 - 1)/ -1), 0) = 10

The following program is an example of DO loop nesting. Two inner DO loops are nested within one outer DO loop.

```
      DO 30 I = 1, 2
      PRINT *, 'OUTER', I
      DO 10 J = 1, 4, 2
      PRINT *, 'INNER J', I, J
   10 CONTINUE
      DO 20 K = 2, 4, 2
      PRINT *, 'INNER K', I, K
   20 CONTINUE
   30 CONTINUE
```

Results from the nested DO example:

```
OUTER        1
INNER J      1    1
INNER J      1    3
INNER K      1    2
INNER K      1    4
OUTER        2
INNER J      2    1
INNER J        2    3
INNER K        2    2
INNER K        2    4
```

The following program fragments show the proper way to use an END DO statement to denote the end of a DO loop.

Without a statement label:

```
      DO I = 1,10
         A(I) = A(I) + 1
      END DO
```

With a statement label:

```
      DO 10 I = 1,10
         A(I) = A(I) + 1
   10 END DO
```

Implied DO in a DATA Statement

The form of an implied DO list in a DATA statement is:

Syntax

(*dlist*, *i* = *m1*, *m2* [, *m3*])

where:

dlist
: is a list of array element names and implied DO lists.

i
: is the name of an integer variable called the implied DO variable.

m1, *m2*, and *m3*
: are each integer constants or names of integer constants, or expressions containing only integer constants or names of integer constants. An expression may contain implied DO variables of other surrounding implied DO lists that have this implied DO list within their ranges (dlist). *m3* is optional; if omitted, it is assumed to be 1, and the preceding comma must also be omitted.

The range of an implied DO list is *dlist*. An iteration count is established from *m1*, *m2*, and *m3* exactly as for a DO-loop, except that the iteration count must be positive.

Upon completion of the implied DO, the implied DO variable is undefined and may not be used until assigned a value in a DATA statement, assignment statement, or READ statement.

Each subscript expression in *dlist* must be an integer constant or an expression containing only integer constants or names of integer constants. The expression may contain implied DO variables of implied DO lists that have the subscript expression within their ranges.

Valid Implied DO Statement:

The following example uses the implied DO to initialize a two-dimensional character array.

```
CHARACTER CHAR1(3,4)
DATA ((CHAR1(I,J), J=1,4), I=1,3)
     /'A','B','C','D','E','F','G','H','I','J','K','L'/
```

The resultant array would be initialized as follows:

Row 1:	A	B	C	D
Row 2:	E	F	G	H
Row 3:	I	J	K	L

Implied DO in an Input/Output Statement

If an implied DO appears in the *list* parameter of an input/output statement, the items specified by the implied DO are transmitted to or from the file. The implied DO list in an input/output statement is of the form:

(*dlist*, *i* = *m1*, *m2* [, *m3*])

where:

dlist
is an input/output list.

i
is the name of an integer, real, or double precision variable (not an array element) called the DO variable.

m1, *m2*, and *m3*
are integer, real, or double precision arithmetic expressions. The values of the expressions *m1*, *m2*, and *m3* are converted to the type of the DO variable *i*, if necessary. *m3* is optional and cannot have a value of 0; if it is omitted, its value is assumed to be 1, and the preceding comma must be omitted.

In an input statement, the DO-variable *i*, or an associated entity, must not appear as an input list item in *dlist*. When an implied-DO list appears in an input/output list, the list items in *dlist* are specified once for each iteration of the implied DO list with appropriate substitution of values for any occurrence of the DO-variable *i*.

For example, assume that A is a variable and that B, C, and D are one-dimensional arrays, each containing 20 elements. Then the statement:

```
READ (UNIT=5)A,B,(C(I),I=1,4),D(4)
```

reads one value into A, the next 20 values into B, and the next 4 values into the first four elements of the array C, and the next value into the fourth element of. D.

Or the statement:

```
WRITE (UNIT=6)A,B,(C(I),I=1,4),D(4)
```

writes one value from A, the next 20 values from B, and the next 4 values from the first four elements of the array C, and the next value from the fourth element of D.

If the subscript (I) were not included with the array C, the entire array would be transferred four times.

Implied DOs can be nested, if required. For example, to read an element into array B after values are read into each row of a 10-by-20 array A, the following input statement would be written:

```
READ (UNIT=5)((A(I,J),J=1,20),B(I),I=1,10)
```

Or, to write an element from array B after values are written into each row of a 10x20 array A, the following output statement would be written:

```
WRITE (UNIT=6)((A(I,J),J=1,20),B(I),I=1,10)
```

The order of the names in the list specifies the order in which the data is to be transferred.

DOUBLE PRECISION Type Statement

See "Explicit Type Statement" on page 91.

DO WHILE Statement

The DO WHILE statement controls the execution of a group of statements that follow it, up to and including the required terminating END DO statement, based on the value of a logical expression. This group of statements is called the "range of the DO WHILE" or a "DO WHILE loop."

Syntax

DO [*stl* [,]] WHILE (*m*)

stl

is the statement label of the END DO statement, in the same program unit as the DO WHILE statement, that denotes the end of the DO WHILE loop. (Whether or not the statement label is used, an END DO statement must be used to denote the end of the DO WHILE loop.)

m

is any logical expression.

The execution of a DO WHILE statement causes the logical expression *m* to be evaluated. If *m* is true, then the statements in the range of the DO WHILE statement are executed and will continue to be executed until *m* is evaluated to be false.

If *m* is initially false, then the range of the DO WHILE is not executed.

No transfers may be made to any of the executable statements within the range of the DO WHILE by statements outside the range of the DO WHILE.

Each DO WHILE loop must have a separate END DO statement.

You can code a DO loop within a DO WHILE loop and a DO WHILE loop within a DO loop. However, you must include the range of the inner loop entirely within the range of the outer loop. The loops may not use the same terminal statement.

Valid DO WHILE statements:

The following program fragment shows how to use a DO WHILE without a label.

```
CONVERGE = .FALSE.
DO WHILE (.NOT. CONVERGE)
   CALL FITTER (X, Y, Z, CONVERGE)
END DO
```

EJECT Statement

EJECT is a compiler directive. It starts a new full page of the source listing. The EJECT statement should not be continued.

Syntax

EJECT

ELSE Statement

See "IF Statements" on page 127.

ELSE IF Statement

See "IF Statements" on page 127.

END Statement

The END statement defines a program unit. That is, it terminates a main program, or a function, subroutine, or block data subprogram.

Syntax

END

The END statement may be numbered. It may not be continued, and no other statement in the program unit may have an initial line that appears to be an END statement. The END statement terminates program execution if it is executed in the main program. If executed in a subprogram, it has the effect of a RETURN statement.

Execution of an END statement terminates the association between the dummy arguments of the subprogram and the current actual arguments. All entities within the subprogram become undefined except:

- Entities specified in SAVE statements. (See "SAVE Statement" on page 203.)
- Entities in a blank common block.
- Initially defined entities that have neither been redefined nor become undefined.
- Entities in named common blocks that appear in the subprogram and appear in at least one other program unit that is referring, either directly or indirectly, to that subprogram. The entities in a named common block may become undefined by execution of a RETURN or END statement in another program unit.

All variables that are assigned a statement label with the ASSIGN statement become undefined, regardless of whether the variable is in a common block or specified in a SAVE statement.

An END statement cannot terminate the range of a DO loop.

END Statement in a Subprogram

All function subprograms must end with END statements. They may also contain RETURN statements. An END statement specifies the physical end of the subprogram.

If the END statement is reached during execution of the subroutine subprogram, it is executed as a RETURN statement.

END DEBUG Statement

The END DEBUG statement terminates the last debug packet for the program.

Syntax

END DEBUG

END DEBUG is placed after the other debug statements and just before the first statement of the program unit being debugged. Only one END DEBUG statement is allowed in a program unit.

See "DEBUG Statement" on page 71.

END DO Statement

The END DO statement may terminate the range of a DO loop and must terminate the range of a DO WHILE loop.

Syntax

END DO

END DO Statement in a DO Loop: The END DO statement must be used if the DO statement that specifies the DO loop does not specify a label. If both a statement label and an END DO are used, the label must be the statement label of the END DO statement.

END DO Statement in a DO WHILE Loop: The END DO statement must be used to terminate a DO WHILE loop. If a statement label is specified on the DO WHILE statement, the label must be the statement label of the END DO statement that terminates the loop.

ENDFILE Statement

The ENDFILE statement writes an end-of-file record on a sequentially accessed external file.

Syntax

ENDFILE *un*

ENDFILE
    ([UNIT=]*un*
    [, ERR=*stl***]**
    [, IOSTAT=*ios***])**

UNIT=*un*

un is the external unit identifier. It is an integer expression of length 4, whose value must be zero or positive. *un* is required.

If the second form of the statement is used, *un* can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the ENDFILE statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the ENDFILE statement.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* value is set positive if an error is detected; it is set to zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*.

Valid ENDFILE Statements:

```
ENDFILE 8

ENDFILE (8,ERR=99999)

ENDFILE (ERR=99999,UNIT=8)
```

Invalid ENDFILE Statements:

```
ENDFILE UNIT=08           UNIT= is not allowed outside parentheses.

ENDFILE 08,ERR=99999      Parentheses must be specified.

ENDFILE (ERR=99999,08)    UNIT= must be specified
                          or un must be first in the list.
```

When the ENDFILE statement is encountered, the unit specified by *un* must be connected to an external file with sequential access. (See *VS FORTRAN Version 2 Programming Guide* for an example.) If the unit is not connected, an error is detected.

When the NOOCSTATUS run-time option is in effect, the unit does not have to be connected. For more information on the NOOCSTATUS option, see *VS FORTRAN Version 2 Programming Guide.*

After successful execution of the ENDFILE statement, the external file connected to the unit specified by *un* is created, if it does not already exist.

Use of ENDFILE with asynchronous READ and WRITE statements is allowed, provided that any input or output operation on the file has been allowed to complete by the execution of a WAIT statement. A WAIT statement is not required to complete the ENDFILE operation.

If an error is detected, transfer is made to the statement specified by the ERR=. If IOSTAT=*ios* is specified, a positive integer value is assigned to *ios* when an error is detected. Execution then continues with the statement designated on the ERR specifier, if present, or with the next statement if ERR is omitted.

END IF Statement

See "IF Statements" on page 127.

ENTRY Statement

The ENTRY statement names the place in a subroutine or function subprogram that can be used in a CALL statement or as a function reference.

The normal entry into a *subroutine* subprogram from the calling program is made by a CALL statement that refers to the subprogram name. The normal entry into a *function* subprogram is made by a function reference in an arithmetic, character, or logical expression. Entry is made at the first executable statement following the SUBROUTINE or FUNCTION statement.

It is also possible to enter a subprogram by a CALL statement (for a *subroutine* subprogram) or a function reference (for a *function* subprogram) that refers to an ENTRY statement in the subprogram. Entry is made at the first executable statement following the ENTRY statement.

Syntax

ENTRY *name* **[(** *arg1* **[,** *arg2* **...])]**

name
: is the name of an entry point in a subroutine or function subprogram. If ENTRY appears in a subroutine subprogram, *name* is a *subroutine name*. If ENTRY appears in a function subprogram, *name* is a *function name*.

arg
: is an optional dummy argument corresponding to an actual argument in a CALL statement or in a function reference. (See "Subprogram Statements" on page 48.) If no *arg* is specified, the parentheses are optional.

arg may be a variable name, array name, or dummy procedure name or an asterisk. An asterisk is permitted only in an ENTRY statement in a *subroutine* subprogram.

The ENTRY statement cannot appear in a main program.

A function subprogram must not refer to itself or to any of its entry points either directly or indirectly.

ENTRY statements are nonexecutable and do not affect control sequencing during execution of a subprogram. They can appear anywhere after a FUNCTION or SUBROUTINE statement, except that an ENTRY statement must not appear between a block IF statement and its matching END IF statement or between a DO statement and the terminal statement of its range.

Note: ENTRY statements can appear before the IMPLICIT or PARAMETER statements. The appearance of an ENTRY statement does not alter the rule that statement functions must precede the first executable statement.

Within a function or subroutine subprogram, an entry name must not appear as a dummy argument of a FUNCTION, SUBROUTINE, or ENTRY statement and it must not appear in an EXTERNAL statement.

If information for an object-time dimension array is passed in a reference to an ENTRY statement, the array name and all its dimension parameters (except any that are in a common area) must appear in the argument list of the ENTRY statement. (See "Size and Type Declaration of an Array" on page 26.)

In a function subprogram, the type of the function name and entry name are determined (in order of decreasing priority) by:

1. An explicit type statement
2. An IMPLICIT statement
3. Predefined convention

In function subprograms, an entry name must not appear preceding the entry statement, except in a type statement.

If any entry name in a function subprogram or the name of the function subprogram is of type character, all entry names of the function subprogram must be of type character with the same length. The CHARACTER type statement or IMPLICIT statement can be used to specify the type and length of the entry point name. The length specification is restricted to the forms permitted in the FUNCTION statement.

The types of these variables (that is, the function name and entry names) can be different only if the type is not character; the variables are treated as if they were equivalenced. After one of these variables is assigned a value in the subprogram, any others of different type become indeterminate in value.

In a function subprogram, either the function name or one of the entry names must be assigned a value.

Upon exit from a function subprogram, the value returned is the value last assigned to the function name or any entry name. It is returned as though it were assigned to the name in the current function reference. If the last value is assigned to a different entry name, and that entry name differs in type from the name in the current function reference, the value of the function is undefined.

Note: Entry names in a subroutine subprogram do not have a type; explicit typing is not allowed.

Valid ENTRY Statements:

To illustrate the use of the ENTRY within a subroutine subprogram, the following subprogram is defined:

```
   SUBROUTINE SAMPLE(A,I,C)
   X = A**I
   GO TO 10
   ENTRY ALIAS(B,C)
   X = B
10 C = SQRT(X)
   RETURN
   END
```

The subprogram invocation

```
CALL SAMPLE(X,J,Z)
```

evaluates the expression SQRT(X**J) and returns the value in Z.

The subprogram invocation

```
CALL ALIAS(Y,W)
```

evaluates the expression SQRT(Y) and returns the value in W.

Actual Arguments in an ENTRY Statement

Entry into a function subprogram associates actual arguments with the dummy arguments of the referenced ENTRY statement. Thus, all appearances of these arguments in the subprogram become associated with actual arguments.

See "Actual Arguments in a Subroutine Subprogram" on page 209 and "Actual Arguments in a Function Subprogram" on page 122.

Dummy Arguments in an ENTRY Statement

The dummy arguments in the ENTRY statement need not agree in order, type, or number with the dummy arguments in the SUBROUTINE or FUNCTION statement or any other ENTRY statement in the same subprogram. However, the actual arguments for each CALL or function reference must agree in order, type, and number with the dummy arguments in the SUBROUTINE, FUNCTION, or ENTRY statement to which it refers.

Unless it has already appeared as a dummy argument in an ENTRY, SUBROUTINE, or FUNCTION statement prior to the executable statement, any dummy argument of an ENTRY statement must not be in an executable statement preceding the ENTRY statement.

If an ENTRY dummy argument is used as an adjustable array name, the array name and all its dimensions (except those in a common block) must be in the same dummy argument list.

Dummy arguments can be variables, arrays, dummy procedure names, or asterisks. The asterisk is allowed only in an ENTRY statement in a subroutine subprogram and indicates an alternate return parameter.

Unless the name is also a dummy argument to the statement function, or is in a FUNCTION or SUBROUTINE statement, or is in an ENTRY statement prior to the statement function definition, a dummy argument must not appear in the expression of a statement function definition.

A dummy argument used in an executable statement is allowed only if that dummy argument appears in the argument list of the FUNCTION, SUBROUTINE, or ENTRY statement by which the subprogram was entered.

See "Dummy Arguments in a Subroutine Subprogram" on page 209 and "Dummy Arguments in a Function Subprogram" on page 123.

EQUIVALENCE Statement

The EQUIVALENCE statement permits the sharing of data storage within a single program unit.

Syntax

EQUIVALENCE (*list1*) [, (*list2*) ...]

list

is a list of variable, array, array element, or character substring names. Names of dummy arguments of an external procedure in a subprogram must not appear in the list. Each pair of parentheses must contain at least two names.

The number of subscript quantities of array elements must be equal to the number of dimensions of the array. If an array name is used without a subscript in the EQUIVALENCE statement, it is interpreted as a reference to the first element of the array.

An array element refers to a position in the array in the same manner as it does in an assignment statement (that is, the array subscript specifies a position relative to the first element of each dimension of the array).

The subscripts and substring information may be integer expressions containing only integer constants, or names of integer constants. They must not contain variables, array elements, or function references.

All the named data within a single set of parentheses shares the same storage location. When the logic of the program permits it, the EQUIVALENCE statement can reduce the number of bytes used by sharing two or more variables of the same type or different noncharacter types.

Both character and noncharacter data types are allowed in an EQUIVALENCE relationship.

The length of the equivalenced entities can be different. Equivalence between variables implies storage sharing.

Mathematical equivalence of variables or array elements is implied only when they are of the same noncharacter type, when they share exactly the same storage, and when the value assigned to the storage is of that type.

Because arrays are stored in a predetermined order, equivalencing two elements of two different arrays implicitly equivalences other elements of the two arrays. The EQUIVALENCE statement must not contradict itself or any previously established equivalences.

Two variables in one common block or in two different common blocks cannot be made equivalent. However, a variable in a program unit can be made equivalent to a variable in a common block. If the variable that is equivalenced to a variable in the common block is an element of an array, the implicit equivalencing of the rest of the elements of the array can extend the size of the common block. The size of the common block cannot be extended so that elements are added ahead of the beginning of the established common block.

For the following examples of the EQUIVALENCE statement, assume these explicit type declarations:

```
COMMON /COM1/ B(50,50), E(50,50)
INTEGER*4 A(10)
REAL*8 C(50), D(10,10,2), F
CHARACTER*4 C1(10), C2(10)
CHARACTER C3
```

Valid EQUIVALENCE Statements:

1. A locally defined variable sharing named common storage.

   ```
   EQUIVALENCE (A(1), E(1,1))
   ```

2. Equivalence a portion of a multi-dimensioned array to a single-dimensioned array.

   ```
   EQUIVALENCE (C(1), B(1,10))
   ```

3. Equivalence a single element of an array to a variable.

   ```
   EQUIVALENCE (D(10,10,2), F)
   ```

4. The first half of a character array is equivalenced to the second half of another character array. Twenty characters (or 5 array elements) are equivalenced.

   ```
   EQUIVALENCE (C1(6), C2(1))
   ```

5. The last character in a character array is equivalenced to a single character.

   ```
   EQUIVALENCE (C3, C1(10)(4: ))
   ```

Character variables may be equivalenced to noncharacter items.

A character array is equivalenced to the second half of an integer array.

```
EQUIVALENCE (C1(1), A(6))
```

Invalid EQUIVALENCE Statement:

Two variables may not be equivalenced when both are in common.

```
EQUIVALENCE (B(1,1), E(1,1))
```

Explicit Type Statement

The explicit type statement allows you to do the following:

- Specify the type and length of variables, arrays, and user-supplied functions.
- Specify the dimensions of an array.
- Assign initial data values for variables and arrays.

The explicit type statement overrides the IMPLICIT statement, which, in turn, overrides the predefined convention for specifying type.

Syntax

type name1 [, *name2* ...]

type

is COMPLEX, INTEGER, LOGICAL, REAL, DOUBLE PRECISION, or CHARACTER[**len*]

where:

len

specifies the length (number of characters between 1 and 32767). It is optional.

Note: The CHARLEN compiler option may be specified to override the maximum length of the character data type to a range of 1 through 32767. The default maximum length remains 500 characters, or whatever length was set at installation time, if the CHARLEN option has not been specified.

The length *len* can be expressed as:

- An unsigned, nonzero, integer constant.
- An expression with a positive value that contains integer constants, names of integer constants enclosed in parentheses, or an asterisk enclosed in parentheses.

The length **len* immediately following the word character is used as the length specification of any name in the statement that has no length specification attached to it. To override a length for a particular name, see the alternative forms of *name* below. If **len* is not specified, it is assumed to be 1.

The comma in character[*len[,]] must not appear if **len* is not specified. It is optional if **len* is specified.

type

is COMPLEX[**len*1], INTEGER[**len*1], LOGICAL[**len*1], or REAL[**len*1]

where:

**len1*

is optional and *len1* represents one of the permissible length specifications for its associated type as described in Figure 7 on page 22.

name

is a variable, array, function name, or dummy procedure name, or the name of a constant. It can have the form:

a[(*dim*)]

or

a[(*dim*)][**len2*]

where:

a
: is a variable, array, function name, or dummy procedure name.

dim
: is optional. *dim* may only be specified for arrays. It is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array in the form:

```
e1:e2
```

or

```
e2
```

where:

e1
: is the lower dimension bound. It is optional. If *e1* (with its following colon) is not specified, its value is assumed to be 1.

e2
: is the upper dimension bound and must always be specified.

(For rules about dimension bounds, see "Size and Type Declaration of an Array" on page 26.)

If a specific intrinsic function name appears in an explicit specification statement that causes a conflict with the type specified for this function in Chapter 5, "Intrinsic Functions" on page 243, the name loses its intrinsic function property in the program unit. A type statement that confirms the type of an intrinsic function is permitted. If a generic function name appears in an explicit specification statement, it does not lose its generic property in the program unit.

**len2*
: overrides the length as specified in the statement by CHARACTER[**len*[,]].

Any length assigned must be an allowable value for the associated variable or array type. The length specified (or assigned by default) with an array name is the length of each element of the array.

If the length specification (*len*) is a constant, it must be an unsigned, nonzero, integer constant. If the length specification is an arithmetic expression enclosed in parentheses, it may contain only integer constants or names of integer constants. Function and array element references must not appear in the expression. The value of the expression must be a positive, nonzero, integer constant.

If the CHARACTER statement is in a main program, and the length of *name* is specified as an asterisk enclosed in parentheses (*)—also known as inherited length—then *name* must be the name of a character constant. The character constant assumes the length of its corresponding expression in a PARAMETER statement.

If the CHARACTER statement is in a subroutine subprogram, and the length of *name* is specified as an asterisk enclosed in parentheses (*), *name* must be the name of a dummy argument or the name of a character constant defined in a PARAMETER statement. The dummy argument assumes the length of the associated actual argument for each reference to the subroutine. The character constant assumes the length of its corresponding expression in a PARAMETER statement.

If the CHARACTER statement is in a function subprogram and the length of *name* is specified as an asterisk enclosed in parentheses (*), *name* must be either the name of a dummy argument, the name of the function in a FUNCTION or ENTRY statement in the same program, or the name of a character constant defined in a PARAMETER statement. If *name* is the name of a dummy argument, the dummy argument assumes the length of the associated actual argument for each reference to the function. If *name* is the function or entry name, when a reference to such a function is executed, the function assumes the length specified in the calling program unit. The character constant assumes the length of its corresponding expression in a PARAMETER statement.

An alternative method of specifying both the length and the type of a function name is by using the FUNCTION statement itself with the optional type declaration (see "FUNCTION Statement" on page 120).

The length of a statement function of character type cannot be specified in the calling program by an asterisk enclosed with parentheses (*), but can be an integer constant expression.

The length specified for a character function in a main program unit that refers to the function must be an expression involving only integer constants or names of integer constants. This length must agree with the length specified in the subprogram that specifies the function, if the length is not specified as an asterisk enclosed with parentheses (*).

name

is a variable, array, function name, or dummy procedure name, or the name of a constant. It can have the form:

a[**len3*][(*dim*)]

or

a[**len3*][(*dim*)] [/*i1*,*i2*,*i3*,.../]

where:

a

is a variable, array, function name, or dummy procedure name.

**len3*

overrides the length as specified in the initial keyword of the statement as COMPLEX, INTEGER, LOGICAL, REAL, COMPLEX[**len1*], CHARACTER[**len*], INTEGER[**len1*], LOGICAL[**len1*], or REAL[**len1*]

dim

is optional. *dim* may only be specified for arrays. It is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array. See the description of *dim* above.

i1,i2,i3,...
are optional and represent initial data values.

Dummy arguments and names of constants, functions, and statement functions may not be assigned initial values.

Initial data values may be assigned for any items of type double precision.

Initial data values may be assigned to variables or arrays that are not dummy arguments or in blank common, by use of *in*, where *in* is a constant or list of constants separated by commas. Each *in* provides initialization only for the immediately preceding variable or array. Lists of constants are used only to assign initial values to array elements. The data must be of the same type as the variable or array, except that hexadecimal data may also be used.

Note: If hexadecimal data is used, the hexadecimal constant form must be followed. (See "Hexadecimal Constants" on page 21.)

Successive occurrences of the same constant can be represented by the form i*constant, as in the DATA statement. If initial data values are assigned to an array in an explicit specification statement, the dimension information for the array must be in the explicit specification statement or in a preceding DIMENSION or COMMON statement.

The following table lists all the possible explicit type statements, and the resulting type and length of the data item.

Type Statement	**Resulting Type**	**Length (Bytes)**
CHARACTER	CHARACTER	1
CHARACTER*n	CHARACTER	n (where 1 ≤ n ≤ x)See note.
COMPLEX	COMPLEX	8
COMPLEX*8	COMPLEX	8
COMPLEX*16	COMPLEX	16
COMPLEX*32	COMPLEX	32
DOUBLE PRECISION	REAL	8
INTEGER	INTEGER	4
INTEGER*2	INTEGER	2
INTEGER*4	INTEGER	4
LOGICAL	LOGICAL	4
LOGICAL*1	LOGICAL	1
LOGICAL*4	LOGICAL	4
REAL	REAL	4
REAL*4	REAL	4
REAL*8	REAL	8
REAL*16	REAL	16

Figure 22. Type and Length of Explicit Type Statements

Note: If the CHARLEN compiler option is not specified, x=500 (or whatever length was specified as the default at installation). If CHARLEN is specified, x=CHARLEN, where x is greater than 0 and less than 32768. For more information about the CHARLEN option, see *VS FORTRAN Version 2 Programming Guide*.

Valid Explicit Type Statements:

```
CHARACTER*8 ORANGE

DATA ORANGE/'ORANGE  '/

SUBROUTINE SUB(DUM)
CHARACTER *(*) DUM

CHARACTER*8 ORANGE/'ORANGE  '/

COMPLEX C,D/(2.1,4.7)/,E*16

INTEGER*2 ITEM/76/, VALUE

REAL A(5,5)/20*6.9E2,4*1.0/,B(100)/100*0.0/,TEST*8(5)/5*0.0D0/

REAL*8 BAKER, HOLD, VALUE*4, ITEM(5,5)
```

EXTERNAL Statement

The EXTERNAL statement identifies a user-supplied subprogram name and permits such a name to be used as an actual argument.

Syntax

EXTERNAL *name1* **[**, *name2* ... **]**

name
: is a name of a user-supplied subprogram (function or subroutine) that is passed as an argument to another subprogram.

EXTERNAL is a specification statement and must precede DATA statements, statement function definitions, and all executable statements.

Statement function names cannot appear in EXTERNAL statements. If the name of a function supplied by VS FORTRAN Version 2 (that is, an intrinsic function) is used in an EXTERNAL statement, the function is no longer recognized as being an intrinsic function when it appears as a function reference. Instead, it is assumed that the function is supplied by the user.

The same name may not appear in both an EXTERNAL and an INTRINSIC statement.

The name of any subprogram that is passed as an argument to another subprogram must appear in an EXTERNAL or INTRINSIC statement in the calling program.

FORMAT Statement

The FORMAT statement is used with the input/output list in the READ and WRITE statements to specify the structure of FORTRAN records and the form of the data fields within the records.

Syntax

FORMAT (*f1* [, *f2*, ...]

f1, *f2*, ... are format codes. The valid format codes are as follows:

Code	Format	Description
I	*a*I*w*	Integer data fields
I	*a*I*w.m*	Integer data fields
D	*a*D*w.d*	Double precision data fields
E	*a*E*w.d*	Real data fields
E	*a*E*w.d*E*e*	Real data fields
F	*a*F*w.d*	Real data fields
G	*a*G*w.d*	Real data fields
G	*a*G*w.d*E*e*	Real data fields
P	*n*P	Scale factor
L	*a*L*w*	Logical data fields
A	*a*A	Character data fields
A	*a*A*w*	Character data fields
	'character constant'	Literal data (character constant)
H	*w*H	Literal data (Hollerith constant)
X	*w*X	Input: Skip a field Output: Fill with blanks
T	T*r*	Transfer of data starts in current position
TL	TL*r*	Transfer of data starts *r* characters to the left of current position
TR	TR*r*	Transfer of data starts *r* characters to the right of current position
group	*a*(...)	Group format specification
S	S	Display of optional plus sign is restored
SP	SP	Plus sign is produced in output
SS	SS	Plus sign is not produced in output
BN	BN	Blanks are ignored in input
BZ	BZ	Blanks are treated as zeros in input
slash	/	Data transfer on the current record is ended
colon	:	Format control is terminated if there are no more items in the input/output list
E	*a*E*w.d*D*e*	Extended precision data fields
G	*a*G*w.d*	Integer or logical data fields
G	*a*G*w.d*E*e*	Integer or logical data fields
Q	*a*Q*w.d*	Extended precision data fields
Z	*a*Z*w*	Hexadecimal data fields

Notes:

a is an optional repeat count—an unsigned, nonzero, integer constant used to denote the number of times the format code or group is to be used. The range of *a* is 1 to 255. If *a* is omitted, the code or group is used only once.

w is an unsigned, nonzero, integer constant that specifies the width of the field. This width must be less than 256.

m is an unsigned integer constant that specifies the number of digits to be printed.

d is an unsigned integer constant that specifies the number of digits to the right of the decimal point:

e is an unsigned, nonzero, integer constant that specifies the number of digits in the exponent field.

n is an (optionally) signed integer constant that specifies a scale factor to be applied.

r is an unsigned, nonzero, integer constant that specifies a character position in a record.

(...) is a group format specification. Within the parentheses are format codes or additional levels of groups, separated by commas, slashes, or colons. Commas are optional before or after a slash and before or after a colon, if the slash or colon is not part of a character constant.

The FORMAT statement is used with READ and WRITE statements for internal and external files. The external files must be connected for SEQUENTIAL or DIRECT access. In the FORMAT statement, the data fields are described with format codes, and the order in which these format codes are specified determines the structure of the FORTRAN records. The I/O list gives the names of the data items that make up the record. The length of the list, in conjunction with the FORMAT statement, specifies the length of the record. (See "Forms of a FORMAT Statement" on page 100.)

The format codes delimited by left and right parentheses may appear as a character constant in the format specification of the READ or WRITE statement, instead of in a separate FORMAT statement. For example,

```
READ (UNIT=5,FMT='(I3,F5.2,E10.3,G10.3)')N,A,B,C

READ (5,'(I3,F5.2,E10.3,G10.3)')N,A,B,C
```

Throughout this section, the examples show 80-column input and printed line output. However, the concepts apply to all input/output media. In the examples, the character ƀ represents a blank.

General Rules for Data Conversion

The following is a list of general rules for using the FORMAT statement or a format in a READ or WRITE statement.

- FORMAT statements are not executed; their function is to supply information to the object program. They may be placed anywhere in a program unit other than in a block data subprogram, subject to the rules for the placement of the PROGRAM, FUNCTION, SUBROUTINE, and END statements.

- Complex data in records requires two successive D, E, G, F, or Q format codes.

  The two codes may be different and the format codes T, TL, TR, X, /, :, S, SP, SS, P, BN, BZ, H, or a character constant may appear between the two codes.

- When defining a record by a FORMAT, it is important to consider the maximum size record allowed on the input/output medium. For example, if a record is to be punched for output, the record should not be longer than 80 characters. If it is to be printed, it should not be longer than the printer's line length.

- When records are to be printed, the first character of each record functions as a carrier control character. The control character determines the vertical spacing of the printed record and is not considered as part of a data item, as follows:

Control Character	***Vertical Spacing Before Printing***
blank	Advance one line.
0	Advance two lines.
1	Advance to first print position on next page.
+	No advance (overstrike).

  The control character is commonly specified in a FORMAT statement, using either of two forms of character constant data, 'x' or 1Hx, where x is one of the control characters shown above. The characters and spacing shown are those defined for print records, and the result of using other characters in the control position is indeterminate (except that the control position is always discarded). If the print record contains no characters, spacing is advanced by one, and a blank line is printed.

  If records are to be displayed at a terminal, control characters are also employed, and characters blank and zero (only) produce the spacing shown above when used in the control position.

  Note: In records that are not to be printed or displayed, the first character of the record is treated as data.

- If the I/O list is omitted from the READ or WRITE statement, the following general rules apply:

  - **Input**: A record is skipped

  - **Output**: A blank record is written unless the FORMAT statement contains an H format code or a character constant (see "H Format Code and Character Constants" on page 111).

    To produce a blank record on output, an empty format specification of the form FORMAT ( ) may be used.

- To illustrate the nesting of group format specifications, the following statements are both valid:

```
FORMAT (...,a(...,a(...),...,a(...),...)
```

  or

```
FORMAT (...,a(...,a(...,a(...),...),...)
```

  where *a* is $1 \leq a < 256$.

- To illustrate the use of nesting in an implied DO and the corresponding FORMAT specifications:

```
      PROGRAM FMT1
      DIMENSION IRR(3,4), IRI(3,4)
      DO 10 I = 1, 3
      DO 10 J = 1, 4
      IRR(I,J) = 1000 + (I * 100) + J
      IRI(I,J) = 2000 + (I * 100) + J
   10 CONTINUE
      PRINT  20, (I, (IRR(I,J), IRI(I,J), J = 1, 4),
     1  I = 1, 3)
   20 FORMAT (3(1X, 'ROW', I3, 4( I5, 1X, I4, 3X) / ))
      STOP
      END
```

Results of program FMT1:

```
ROW  1 1101 2101    1102 2102    1103 2103    1104 2104
ROW  2 1201 2201    1202 2202    1203 2203    1204 2204
ROW  3 1301 2301    1302 2302    1303 2303    1304 2304
```

- Names of constants must not be a part of a format specification (see "PARAMETER Statement" on page 158).

- If you use the comma as an input delimiter with the format codes I, D, E, F, G, L, Q, and Z, you do not need to align the data with leading zeroes or blanks. The comma will override the format specifications when the comma appears before the end of the field width. Two successive commas or a comma immediately after a field indicate that the data should be considered FALSE for format code L and 0 for codes I, D, E, F, G, Q, and Z.

  For example, with the program fragment:

  ```
        READ (10,3000) A, C, B, II, X
   3000 FORMAT (F7.2, A8, F8.5, I10, E10.5)
  ```

  And the following input:

  ```
  1,abcdefgh,,32,1,7
  ```

  The following assignments would occur:

  ```
  A = .01         (not 1.00)
  C = abcdefgh    (the format A is not controlled by the comma)
  B = 0.0         (because of the two consecutive commas)
  II = 32
  X = .00001      (not 1.00000, since no decimal was used in the input)
  ```

  And the ,7 would be ignored.

- With numeric data format codes I, F, E, G, and D, the following general rules apply:

  - **Input**: Leading blanks are not significant. The interpretation of blanks, other than leading blanks, is determined by a combination of the value of the BLANK= specifier given when the file was connected (see "OPEN Statement" on page 151) and any BN or BZ blank control that is currently in effect. Plus signs may be omitted. A field of all blanks is considered to be zero.

    With F, E, G, and D format codes, a decimal point appearing in the input field overrides the portion of a format code that specifies the decimal point location. The input field may have more digits than VS FORTRAN Version 2 uses to approximate the value.

- **Output**: The representation of a positive or zero internal value in the field may be prefixed with a plus, as controlled by the S, SP, and SS format codes. The representation of a negative internal value in the field is prefixed with a minus. A negative zero is not produced.

  The representation is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks are inserted in the field.

  If the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w*.*d*E*e* or G*w*.*d*E*e* format codes, the entire field of width *w* is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, asterisks are not produced. When an SP format code is in effect, a plus is not optional.

With Q editing and D exponents, the following additional rules apply:

- **Input**: With Q editing, a decimal point appearing in the input field overrides the portion of a format code that specifies the decimal point location. The input field may have more digits than VS FORTRAN Version 2 uses to approximate the value.

- **Output**: If the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w*.*d*D*e* or Q*w*.*d* format codes, the entire field of width *w* is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, asterisks are not produced. When an SP format code is in effect, a plus is not optional.

Forms of a FORMAT Statement

All the format codes in a FORMAT statement are enclosed in parentheses. Within these parentheses, the format codes are delimited by commas. The comma used to separate list items may be omitted as follows:

- Between a P edit descriptor and an immediately following F, E, D, or G format code
- Before or after a slash format code
- Before or after a colon format code

Execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information provided jointly by the I/O list, if one exists, and the format specification. If there is an I/O list, there must be at least one I, D, E, F, A, G, L, Q, or Z format code in the format specification.

There is no I/O list item corresponding to the format codes: T, TL, TR, X, H, character constants enclosed in apostrophes, S, SP, SS, BN, BZ, P, the slash (/), or the colon (:). These communicate information directly to the record.

Whenever an I, D, E, F, A, G, L, Q or Z format code is encountered, format control determines whether there is a corresponding element in the I/O list.

If there is a corresponding element, appropriately converted information is transmitted. If there is no corresponding element, the format control terminates, even if there is an unsatisfied repeat count.

When format control reaches the last (outer) right parenthesis of the format specification, a test is made to determine whether another element is specified in the I/O list. If not, control terminates. If another list element is specified, the format control starts a new record. Control then reverts to that group specification terminated by the last preceding right parenthesis, including its group repeat count, if any, or, if no group specification exists, then to the first left parenthesis of the format specification. Such a group specification must include a closing right parenthesis. If no group specification exists, control reverts to the first left parenthesis of the format specification.

For example, assume the following FORMAT statements:

```
70 FORMAT (I5,2(I3,F5.2),I4,F3.1)

80 FORMAT (I3,F5.2,2(I3,2F3.1))

90 FORMAT (I3,F5.2,2I4,5F3.1)
```

With additional elements in the I/O list after control has reached the last right parenthesis of each, control would revert to the 2(I3,F5.2) specification in the case of statement 70; to 2(I3,2F3.1) in the case of statement 80; and to the beginning of the format specification, I3,F5.2,... in the case of statement 90.

The question of whether there are further elements in the I/O list is asked only when an I, D, E, F, A, G, L, Q, or Z format code or the final right parenthesis of the format specification is encountered.

Before this is done, T, TL, TR, X, and H codes, character constants enclosed in apostrophes, colons, and slashes are processed. If there are fewer elements in the I/O list than there are format codes, the remaining format codes are ignored.

I Format Code

The I format code edits integer data. For example, if a READ statement refers to a FORMAT statement containing I format codes, the input data is stored in internal storage in integer format. The magnitude of the data to be transmitted must not exceed the maximum magnitude of an integer constant.

Input: Leading blanks in a field of the input line are interpreted as zeros. Embedded and trailing blanks are treated as indicated in the general rules for numeric fields described under "General Rules for Data Conversion" on page 97. If the form I*w*.*m* is used, the value of *m* has no effect.

Output: The output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise. If the number of significant digits and sign required to represent the quantity in the datum is less than *w*, the unused leftmost print positions are filled with blanks. If it is greater than *w*, asterisks are printed instead of the number. If the form I*w*.*m* is used, the output is the same as the I*w* form, except that the unsigned integer constant consists of at least *m* digits and, if necessary, has leading zeros. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

F Format Code

The F*w*.*d* format code edits real data. It indicates that the field occupies *w* positions, the fractional part of which consists of *d* digits.

Input: The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented.

The input field may have more digits than VS FORTRAN Version 2 uses to approximate the value of the datum. The basic form may be followed by an exponent of one of the following forms:

- Signed integer constant
- E followed by zero or more blanks, followed by an optionally signed integer constant
- D followed by zero or more blanks, followed by an optionally signed integer constant
- Q followed by zero or more blanks, followed by an optionally signed integer constant

An exponent containing a D or Q is processed identically to an exponent containing an E.

Output: The output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise. This is followed by a string of digits that contains a decimal point, representing the magnitude of the internal value, as modified by the established scale factor and rounded to *d* fractional digits. Leading zeros are not provided, except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero also appears if there would otherwise be no digits in the output field.

D, E, and Q Format Codes

The D*w*.*d*, E*w*.*d*, E*w*.*d*E*e* format codes edit real, complex, or double precision data.

The E*w*.*d*D*e* and Q*w*.*d* format codes edit **extended precision data** in addition to real, complex, and double precision data.

The external field occupies *w* positions, the fractional part of which consists of *d* digits (unless a scale factor greater than 1 is in effect). The exponent part consists of *e* digits. (The *e* has no effect on input.)

Input: The input field may have more digits than VS FORTRAN Version 2 uses to approximate the value of the datum.

Input datum must be a number, which, optionally, may have a D, E, or Q exponent, or which may be omitted from the exponent if the exponent is signed.

All exponents must be preceded by a constant; that is, an optional sign followed by at least one decimal digit with or without decimal point. If the decimal point is present, its position overrides the position indicated by the *d* portion of the format code, and the number of positions specified by *w* must include a place for it. If the data has an exponent, and a P format code is in effect, the scale factor is ignored.

The interpretation of blanks is explained in "General Rules for Data Conversion" on page 97.

The input datum may have an exponent of any form. The input datum is converted to the length of the variable as specified in the I/O list. The *e* of the exponent in the format code has no effect on input.

Output: For data written under a D or E format code, unless a P-scale factor is in effect, output consists of an optional sign (required for negative values), an optional zero digit, a decimal point, the number of significant digits specified by *d*, and a D or E exponent requiring four positions.

If the P-scale factor is negative, output consists of an optional sign (required for negative values), an optional zero digit, a decimal point, |***P***| leading zeros, |***d+P***| significant digits, and a D or E exponent requiring four positions. (***P*** is the value of the P-scale factor.)

If the P-scale factor is positive, output consists of an optional sign (required for negative values), ***P*** decimal digits, a decimal point, ***d-P+1*** fractional digits, and a D or E exponent requiring four positions. (***P*** is the value of the P-scale factor.)

For data written under a Q format code, unless a P-scale factor is in effect, output consists of an optional sign (required for negative values), a decimal point, the number of significant digits specified by *d*, and a Q exponent requiring four positions.

On output, *w* must provide sufficient space for an integer segment if it is other than zero, a fractional segment containing *d* digits, a decimal point, and, if the output value is negative, a sign. If insufficient space is provided for the integer portion, including the decimal point and sign (if any), asterisks are written instead of data. If excess space is provided, the number is preceded by blanks.

The fractional segment is rounded to *d* digits. If the output field consists only of a fractional segment, and if additional space is available, a 0 is placed to the left of the decimal point. If the entire value is zero, a 0 is printed before the decimal point.

G Format Code

The G format code is a generalized code used to **transmit real data** according to the type specification of the corresponding variable in the I/O list. The G*w*.*d* and G*w*.*d*E*e* edit descriptors indicate that the external field occupies *w* positions. Unless a scale factor greater than one is in effect, the fractional part of *w* consists of *d* digits. The exponent part consists of *e* digits.

Input: The form of the input field is the same as for the F format code.

Output: The method of representation in the output field depends on the magnitude of the data being edited.

For example, letting N be the magnitude of the internal data,

```
if N < 0.1  or  N ≥ 10**d
```

(where *k* is the scale factor currently in effect), then:

- G*w*.*d* output editing is the same as *k*PE*w*.*d* output editing.
- G*w*.*d*E*e* output editing is the same as *k*PE*w*.*d*E*e* output editing.

If N is greater than or equal to 0.1 and less than 10**d, the scale factor has no effect, and the value of N determines the editing as follows:

Magnitude of Data	Equivalent Conversion
0.1 ≤ N < 1	F(*w*-*n*).*d*, *n* ('ƀ')
1 ≤ N < 10	F(*w*-*n*).(*d*-1), *n* ('ƀ')
⋮	⋮
10**(d-2) ≤ N < 10**(d-1)	F(*w*-*n*).1, *n* ('ƀ')
10**(d-1) ≤ N < 10**d	F(*w*-*n*).0, *n* ('ƀ')

ƀ means blank.

n means:

- 4 for G*w*.*d*
- *e*+2 for G*w*.*d*E*e*

The scale factor has no effect unless the magnitude of the data to be edited is outside the range that permits effective use of F editing.

The letter Q is used for the exponent of extended precision data.

The G format code may be used to transmit integer or logical data according to the type specification of the corresponding variable in the I/O list.

If the variable in the I/O list is integer or logical, the *d* portion of the format code, specifying the number of significant digits, can be omitted; if it is given, it is ignored.

P Format Code

A P format code **specifies a scale factor** *n*, where *n* is an optionally signed integer constant. The value of the scale factor is zero at the beginning of execution of each input/output statement. It applies to all subsequently interpreted F, E, D, G, and Q format codes until another scale factor is encountered; then that scale factor is established.

Reversion of format control does not affect the established scale factor. A repetition code can precede these format codes. For example, 2P,3F7.4 is valid. (A comma must be placed after the P format code—for example, 2P,3F7.4, when a repeat count is specified.) A scale factor of zero may be specified.

Input: If an exponent is in the data field, the scale factor has no effect. If no exponent is in the field, the externally represented number equals the internally represented number multiplied by 10**n for the external representation.

For example, if the input data is in the form

```
xx.xxxx
```

and is to be used internally in the form

```
  .xxxxxx
```

then the format code used to effect this change is

```
2PF7.4
```

which may also be written 2P,F7.4.

Similarly, if the input data is in the form

```
  xx.xxxx
```

and is to be used internally in the form

```
xxxx.xx
```

then the format code used to effect this change is

```
-2PF7.4
```

which may also be written -2P,F7.4.

Output: With an F format code, the internally represented number reduced by 10**n is produced.

For example, if the number has the internal form

```
  .xxxxxx
```

and is to be written in the form

```
xx.xxxx
```

the format code used to effect this change is

```
2PF7.4
```

which may also be written 2P,F7.4.

On output with E, D, and Q format codes, the value of the internally represented number is not changed. When the decimal point is moved according to the *d* of the format code, the exponent is adjusted so that the value of the externally represented number is not multiplied by 10***n*.

For example, if the internal number

```
238.47
```

were printed according to the format E10.3, it would appear as

```
0.238E+03
```

If it were printed according to the format 1PE10.3 or 1P,E10.3, it would appear as

```
2.385E+02
```

On output with a G format code, the effect of the scale factor is suspended unless the magnitude of the internally represented number (*m*) is outside the range that permits the use of F format code editing. This range for use of the F format code is

$$.1 \le m < 10 ** d$$

where *d* is the number of digits as specified in the G format code G*w.d*.

If $.1 \leq m < 10**d$ and the F format code is used, there is **no** difference between G format code **with** a scale factor and G format code **without** a scale factor.

However, if $m \geq 10**d$ or < 0.1, the scale factor moves the decimal point to the right or left.

The following example illustrates the difference between G format code with and without a scale factor:

If A is initially set to 100 and multiplied by 10 each time, and:

```
76  FORMAT ('  ',G13.5,1PG13.5,2PG13.5)
    WRITE  (6,76) A,A,A
```

the result is:

No Scale Factor	Scale Factor = 1	Scale Factor = 2
100.00	100.00	100.00
1000.0	1000.0	1000.0
10000.	10000.	10000.
0.10000E+06	1.00000E+05	10.0000E+04
0.10000E+07	1.00000E+06	10.0000E+05

Z Format Code

The Z format code **transmits hexadecimal data**.

Input: Scanning of the input field proceeds from right to left. Leading, embedded, and trailing blanks in the field are treated as zeros. One byte in internal storage contains two hexadecimal digits; thus, if an input field contains an odd number of digits, the number is padded on the left with a hexadecimal zero when it is stored. If the storage area is too small for the input data, the data is truncated and high-order digits are lost.

Example:

For the following program fragment:

```
       INTEGER*2 INPUT1
       INTEGER*4 INPUT2,INPUT3
       READ (10,1000) INPUT1
       READ (10,1000) INPUT2
       READ (10,1000) INPUT3
1000   FORMAT (Z8)
```

and the following input:

```
AA11CC33
11223344
DD22
```

the variables would receive these values (in hexadecimal):

```
INPUT1 = CC33      (value is truncated to fit in the INTEGER*2 variable)
INPUT2 = 11223344  (input's length the same as variable's length)
INPUT3 = DD220000  (value is padded with zeros on the left)
```

Output: If the number of digits in the datum is less than *w*, the leftmost print positions are filled with blanks. If the number of digits in the byte is greater than *w*, the leftmost digits are truncated and the rest of the number is printed.

Example:

With the following program fragment:

```
      INTEGER*4 OUTPUTA, OUTPUTB, OUTPUTC
      WRITE (11,2000) OUTPUTA
      WRITE (11,2000) OUTPUTB
2000  FORMAT (Z8)
      WRITE (11,3000) OUTPUTC
3000  FORMAT (Z4)
```

and the hexadecimal values in the variables:

```
OUTPUTA = 11BB99AA
OUTPUTB = 3355
OUTPUTC = CC22DD44
```

the following would be the output:

```
11BB99AA    (output and the variable are the same length)
ƀƀƀƀ3355    (output is padded with spaces on the left
DD44        (leftmost digits of the output are truncated)
```

Numeric Format Code Examples

Example 1:

The following example illustrates the use of format codes I, F, D, E, and G.

```
75 FORMAT (I3,F5.2,E10.3,G10.3)

   READ (5,75) N,A,B,C
```

Explanation

- Four input fields are described in the FORMAT statement and four variables are in the I/O list. Therefore, each time the READ statement is executed, one input line is read from the file connected to unit number 5.
- When an input line is read, the number in the first field of the line (three columns) is stored in integer format in location N. The number in the second field of the input line (five columns) is stored in real format in location A, and so on.
- If there were one more variable in the I/O list, for example, M, another line would be read and the information in the first three columns of that line would be stored in integer format in location M. The rest of the line would be ignored.
- If there were one fewer variable in the list (for example, if C were omitted), format code G10.3 would be ignored.
- This FORMAT statement defines only one record format. "Forms of a FORMAT Statement" on page 100 explains how to define more than one record format in a FORMAT statement.

Example 2:

This example illustrates the use of the Z, D, and G format codes.

Assume that the following statements are given:

```
75 FORMAT (Z4,D10.3,2G10.3)

   READ (5,75) A,B,C,D
```

where A, C, and D are REAL*4 and B is REAL*8.
Also, assume that on successive executions of the READ statement, the following input lines are read:

```
Column:   1   5         15        25        35

          v   v         v         v         v
          ƀ3F1156432D+02276.38E+15ƀƀƀƀƀƀƀƀƀƀ
Input
          2AF3155381+02ƀ382506E+28276.38E+15
Lines
          3ACƀ346.18D-03485.322836276.38E+15

Format:   Z4    D10.3     G10.3     G10.3
```

Then ƀ represents a blank and the variables A, B, C, and D receive values as if the following data fields had been supplied:

A	B	C	D
03F1	156.432D02	276.38E+15	000000.000
2AF3	155.381+20	382.506E+28	276.38E+15
3AC0	346.18D-03	485.322836	276.38E+15

Explanation

- Leading blanks in an input field are treated as zeros. If it is assumed that all other blanks are to be treated as zeros, because the value for B on the second input line was not right justified in the field, the exponent is 20, not 2.
- Values read into the variables C and D with a G format code are converted according to the type of the corresponding variable in the I/O list.

Example 3:

This example illustrates the use of the character constant enclosed in apostrophes and the F, E, G, and I format codes.

Assume that the following statements are given:

```
76 FORMAT   ('0',F6.2,E12.3,G14.6,I5)

   WRITE    (6,76)A,B,C,N
```

and that the variables A, B, C, and N have the following values on successive executions of the WRITE statement:

```
  A          B                   C            N

 034.40    123.380E+02       123.380E+02    031

  031.1     1156.1E+02         123456789.   130

-354.32    834.621E-03         1234.56789   428

 01.132     83.121E+06       123380.D+02    000
```

Then, the following lines are printed by successive executions of the WRITE statement:

```
Print
Column:     1       9           21            35

            v       v           v             v

             34.40   0.123E+05   12338.0        31

             31.10   0.116E+06  0.123457E 09   130

            ******    0.835E+00   1234.57       428

              1.13   0.831E+08  0.123380E 08     0
```

Explanation

- The integer portion of the third value of A exceeds the format code specification, so asterisks are printed instead of a value. The fractional portion of the fourth value of A exceeds the format code specification, so the fractional portion is rounded.
- For the variable B, the decimal point is printed to the left of the first significant digit and only three significant digits are printed because of the format code E12.3. Excess digits are rounded off from the right.
- The values of the variable C are printed according to the format specification G14.6. The *d* specification, which in this case is 6, determines the number of digits to be printed and whether the number should be printed with a decimal exponent. Values greater than or equal to 0.1 and less than 1000000 are printed without a decimal exponent in this example. Thus, the first and third values have no exponent. The second and fourth values are greater than 1000000, so they are printed with an exponent.

L Format Code

The L format code **transmits logical variables.**

Input: The input field must consist of either zeros or blanks with an optional decimal point, followed by a T or an F, followed by optional characters, for true and false, respectively. The T or F assigns a value of true or false to the logical variable in the input list. The logical constants .TRUE. and .FALSE. are acceptable input forms.

Output: A T or an F is inserted in the output record, depending upon whether the value of the logical variable in the I/O list was true or false, respectively. The single character is right justified in the output field and preceded by *w*-1 blanks.

A Format Code

The A format code **transmits character data**. Each character is transmitted without conversion.

If *w* is specified, the field consists of *w* characters.

If the number of characters *w* is not specified with the format code A, the number of characters in the field is the length of the character item in input/output list.

Input: The maximum number of characters stored in internal storage depends on the length of the variable in the I/O list. If *w* is greater than the variable length, for example, *v*, then the leftmost *w-v* characters in the field of the input line are skipped, and remaining *v* characters are read and stored in the variable. If *w* is less than *v*, then *w* characters from the field in the input line are read, and remaining rightmost characters in the variable are filled with blanks.

Output: If *w* is greater than the length *v* of the variable in the I/O list, then the printed field contains *v* characters, right-justified in the field, preceded by leading blanks. If *w* is less than *v*, the leftmost *w* characters from the variable are printed, and the rest of the data is truncated.

Example 1:

Assume that *B* has been specified as CHARACTER*8, that N and M are CHARACTER*4, and that the following statements are given:

```
25   FORMAT (3A7)

     READ   (5,25) B, N, M
```

When the READ statement is executed, one input line is read from the data set associated with data set reference number 5 into the variables B, N, and M, in the format specified by FORMAT statement 25. The following list shows the values stored for the given input lines (ƀ represents a blank).

Input Line	B	N	M
ABCDEFG46ƀATƀ11234567	ABCDEFGƀ	ATƀ1	4567
HIJKLMN76543213334445	HIJKLMNƀ	4321	4445

Example 2:

Assume that A and B are character variables of length 4, that C is a character variable of length 8, and that the following statements are given:

```
26   FORMAT   (A6,A5,A6)

     WRITE    (6,26) A,B,C
```

When the WRITE statement is executed, one line is written on the data set associated with data set reference number 6 from the variables A, B, and C in the format specified by FORMAT statement 26. The printed output for values of A, B, and C is as follows (ƀ represents a blank):

A	B	C	Printed Line

```
A1B2  C3D4   E5F6G7H8  ƀƀA1B2ƀC3D4E5F6G7
```

H Format Code and Character Constants

Character constants can appear in a FORMAT statement in one of two ways: following the H format code or enclosed in apostrophes. For example, the following FORMAT statements are equivalent.

```
25   FORMAT (22H 1982 INVENTORY REPORT)

25   FORMAT (' 1982 INVENTORY REPORT')
```

No item in the output list corresponds to the character constant. The constant is written directly from the FORMAT statement. (The FORMAT statement can contain other types of format code with corresponding variables in the I/O list.)

Input: Character constants cannot appear in a format used for input.

Output: The character constant from the FORMAT statement is written on the output file. (If the H format code is used, the *w* characters following the H are written. If apostrophes are used, the characters enclosed in apostrophes are written.) For example, the following statements:

```
8 FORMAT (14H0MEAN AVERAGE:, F8.4)

  WRITE (6,8) AVRGE
```

would write the following record if the value of AVRGE were 12.3456:

```
MEAN AVERAGE: 12.3456
```

The first character of the output data record in this example is the carrier control character for printed output. One line is skipped before printing, and the carrier control character does not appear in the printed line.

Note: If the character constant is enclosed in apostrophes, an apostrophe character in the data is represented by two successive apostrophes. For example, DON'T would be represented as 'DON''T'. The two successive apostrophes are counted as one character. A maximum of 255 characters can be specified in a character or a Hollerith constant.

X Format Code

The X format code **specifies a field of *w* characters to be skipped** on input or filled with blanks on output if the field was not previously filled. On output, an X format code does not affect the length of a record. For example, the following statements:

- Read the first ten characters of the input line into variable I,
- Skip over the next ten characters without transmission, and
- Read the next four fields of ten characters each into the variables J, K, L, and M.

```
5   FORMAT (I10,10X,4I10)

    READ   (5,5) I,J,K,L,M
```

T Format Code

The T format code **specifies the position in the record at which the transfer of data is to begin.**

To illustrate the use of the T code, the following statements:

```
5   FORMAT (T40,'1981 STATISTICAL REPORT', T80,

  X  'DECEMBER',T1,'0PART NO. 10095')

    WRITE  (6,5)
```

print the following:

```
Print
Position:  1                 39                     79

           v                 v                      v
           PART NO. 10095    1981 STATISTICAL REPORT  DECEMBER
```

The T format code can be used in a FORMAT statement with any type of format code, as, for example, with FORMAT ('0',T40,I5).

Input: The T format code allows portions of a record to be processed more than once, possibly with different format codes.

Output: The record is assumed to be initially filled with blank characters, and the T format code can replace or skip characters. On output, a T format code does not affect the length of a record.

(For printed output, the first character of the output data record is a carrier control character and is not printed. Thus, for example, if T50,'Z' is specified in a FORMAT statement, a Z will be the 50th character of the output record, but it will appear in the 49th print position.)

TL and TR Format Codes: The TL and TR format codes specify how many characters left (TL) or right (TR) from the current character position the transfer of data is to begin. With TL format code, if the current position is less than or equal to the position specified with TL, the next character transmitted will be placed in position 1 (that is, the carrier control position).

The TL and TR format codes can be used in a FORMAT statement with any type of format code. On output, these format codes do not affect the length of a record.

Group Format Specification

The group format specification **repeats a set of format codes and controls the order in which the format codes are used.**

The group repeat count *a* is the same as the repeat indicator *a* that can be placed in front of other format codes. For example, the following statements are equivalent:

```
10   FORMAT  (I3,2(I4,I5),I6)

10   FORMAT  (I3,(I4,I5,I4,I5),I6)
```

Because control returns to the last group repeat specification when there are more items in the I/O list than there are format codes in the FORMAT statement, group repeat specifications control the order in which format codes are used. (See "Forms of a FORMAT Statement" on page 100.) Thus, in the previous example, if there were more than six items in the I/O list, control would return to the group repeat count 2, which precedes the specification (I4,I5).

If the group repeat count is omitted, a count of 1 is assumed. For example, the statements:

```
15   FORMAT  (I3,(F6.2,D10.3))

     READ   (5,15) N,A,B,C,D,E
```

read values from the first record for N, A, and B, according to the format codes I3, F6.2, and D10.3, respectively. Then, because the I/O list is not exhausted, control returns to the last group repeat specification, the next record is read, and values are transmitted to C and D according to the format codes F6.2 and D10.3, respectively. Because the I/O list is still not exhausted, another record is read and value is transmitted to E according to the format code F6.2—the format code D10.3 is not used.

All format codes can appear within the group repeat specification. For example, the following statement is valid:

```
40   FORMAT  (2I3/(3F6.2,F6.3/D10.3,3D10.2))
```

The first physical record, containing two data items, is transmitted according to the specification 2I3; the second, fourth, and so on, records, each containing four data items, are transmitted according to the specification 3F6.2,F6.3; and the third, fifth, and so on, records, each also containing four data items, are transmitted according to the specification D10.3,3D10.2, until the I/O list is exhausted.

S, SP, and SS Format Codes

The S, SP, and SS format codes **control optional plus sign characters in numeric output fields**. At the beginning of execution of each formatted output statement, a plus sign is produced in numeric output fields. If an SP format code is encountered in a format specification, a plus sign is produced in *any subsequent* position that normally contains an optional plus sign. If SS is encountered, a plus sign is not produced in *any subsequent* position that normally contains an optional plus sign. If an S is encountered, the option of producing the plus sign is set off.

Example:

The following program:

```
    DOUBLE PRECISION A
    REAL*16 S
    R=3.
    S=4.
    I=5
    A=1.
    T=7.
    U=8.
    WRITE (6,100) R,S,I,A,T,U
100 FORMAT (F10.2,SP,Q15.3,SS,I7,SP,D10.2,S,E10.3,SP,G10.1)
    STOP
    END
```

produces the following output:

```
3.00    +0.400Q+01    5 +0.10D+01 0.700E+01    +8.
```

The S, SP, and SS format codes affect only I, F, E, G, and D editing during the execution of an output statement.

The S, SP, and SS format codes also affect Q editing.

The S, SP, and SS format codes have no effect during the execution of an input statement.

BN Format Code

The BN format code **specifies the interpretation of blanks**, other than leading blanks, in numeric input fields. At the beginning of each formatted input statement, such blank characters are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier given when the unit was connected. (See "OPEN Statement" on page 151.)

If BN is encountered in a format specification, all such blank characters in *succeeding* numeric input fields are ignored. However, a field of all blanks has the value zero.

The BN format code affects only I, F, E, G, and D editing during execution of an input statement.

The BN format code also affects Q editing during execution of an input statement.

The BN format code has no effect during execution of an output statement.

BZ Format Code

The BZ format code **specifies the interpretation of blanks**, other than leading blanks, in numeric input fields.

If BZ is encountered in a format specification, all non-leading blank characters in *succeeding* numeric fields are treated as zeros. If no OPEN statement is given and the file is preconnected, all non-leading blanks in numeric fields are interpreted as zeros.

The BZ format code affects only I, F, E, G, D, and Q editing during execution of an input statement.

The BZ format code has no effect during execution of an output statement.

Example:

The following program (containing both BN and BZ format codes):

```
      READ   (9,100) R,S,I,J
      REWIND 9
      READ   (9,101) A,B,K,L
100   FORMAT (BZ,E6.3,F7.2,I3,I4)
101   FORMAT (BN,E6.3,F7.2,I3,I4)
      WRITE  (*,102) R,S,I,J
      WRITE  (*,102) A,B,K,L
102   FORMAT (2X,E10.3,F7.2,I7,I4)
      STOP
      END
```

with the following input:

```
Column:  12345678901234567890

         123     315  3     5
```

creates the following output:

```
0.123E+03 315.00    300    5
0.123E+00   3.15      3    5
```

Slash Format Code

A slash **indicates the end of a record**.

On input from a file connected for sequential access, the remaining portion of the current record is skipped, and the file is positioned at the beginning of the next record.

On output to a file connected for sequential access, a new record is created. For example, on output, the statement:

```
25    FORMAT     (I3,F6.2/D10.3,F6.2)
```

describes two record formats. The first, third, and so on, records are transmitted according to the format I3, F6.2 and the second, fourth, and so on, records are transmitted according to the format D10.3, F6.2.

Consecutive slashes can be used to introduce blank output records or to skip input records. If there are *n* consecutive slashes at the beginning or end of a FORMAT statement, *n* input records are skipped or *n* blank records are inserted between output records. If *n* consecutive slashes appear anywhere else in a FORMAT statement, the number of records skipped or blank records inserted is *n*-1. For example, the statement:

```
25    FORMAT     (1X,10I5//1X,8E14.5)
```

describes three record formats. On output, it places a blank line between the line written with format 1X,10I5 and the line written with the format 1X,8E14.5.

For a file connected for direct access, when a slash is encountered, the record number is increased by one and the file is positioned at the beginning of the record that has that record number.

Colon Format Code

A colon **terminates format control** if there are no more items in the input/output list. The colon has no effect if there are more items in the input/output list.

Example:

Assume the following statements:

```
     ITABLE=10
     IELEM=0
       .
       .
       .
  10 WRITE(6,1000)ITABLE,IELEM
       .
       .
       .
     ITABLE=11
     IELEM=25
       .
       .
       .
     XMIN=-.37E1
     XMAX=.2495E3
       .
       .
       .
  20 WRITE(6,1000)ITABLE,IELEM,XMIN,XMAX
1000 FORMAT(' 0TABLE NUMBER',I3,:,' CONTAINS',I3,' ELEMENTS',:,
    1      /' MINIMUM VALUE:',E15.7,
    2      /' MAXIMUM VALUE:',E15.7)
```

The WRITE statement at statement 10 generates the following:

```
TABLE NUMBER 10 CONTAINS  0 ELEMENTS
```

The WRITE statement at statement 20 generates the following:

```
TABLE NUMBER 11 CONTAINS 25 ELEMENTS
MINIMUM VALUE: -.3700000E+01
MAXIMUM VALUE:  .2495000E+03
```

Providing the Format in a Character String

VS FORTRAN Version 2 provides for variable FORMAT statements by allowing a format specification to be read into a character array element or a character variable in storage. The data in the character array or variable may then be used as the format specification for subsequent input/output operations. The format specification may also be placed into the character array or variable by a DATA statement or an assignment statement in the source program. The following rules apply:

- The format specification must be a character array or a character variable.
- The format codes entered into the array or character variable must have the same form as a source program FORMAT statement, except that the

word FORMAT and the statement label are omitted. The parentheses surrounding the format codes are required.

- If a format code read at object time contains two consecutive apostrophes within a character field that is defined by apostrophes, it should be used for output only.
- Blank characters may precede the format specification, and character data may follow the right parenthesis that ends the format specification.

Example:

Assume the following statements:

```
  DIMENSION C(5)
  CHARACTER*16 FMT
  FMT='(2E10.3,5F10.8)'
  READ(5,FMT)A,B,(C(I),I=1,5)
```

The data is read, converted, and stored in A, B, and the array C, according to the format codes 2E10.3, 5F10.8.

Reading a FORMAT into a noncharacter array: Assume the following statements:

```
  DIMENSION RFMT(16),C(5)
  READ(5,1) RFMT
1 FORMAT(16A4)
  READ(5,RFMT)A,B,(C(I),I=1,5)
```

Assume also that the first input line associated with unit 5 contains:

```
(2E10.3,5F10.8)
```

The data on the next input record is read, converted, and stored in A, B, and the array C, according to the format codes 2E10.3, 5F10.8.

List-Directed Formatting

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a constant, a null value, or one of the forms:

```
r*f
```

or

```
r*
```

where _r_ is an unsigned, nonzero, integer constant. The _r*f_ form is equivalent to _r_ successive appearances of the constant _f_, and the _r*_ form is equivalent to _r_ successive null values. Neither of these forms may contain embedded blanks, except where permitted within the constant _f_.

A *value separator* is one of the following:

- A comma, optionally preceded by one or more blanks and optionally followed by one or more blanks
- A slash, optionally preceded by one or more blanks and optionally followed by one or more blanks
- One or more blanks between two constants or following the last constant

Input: Input forms acceptable to format specifications for a given type are acceptable for list-directed formatting, except as noted below. The form of the input value must be acceptable for the type of the input list item. Blanks are never treated as zeros; and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below. The end of a record has the effect of a blank, except when it appears within a character constant.

When the corresponding input list item is of real or double precision type, the input form is that of a numeric input field. A *numeric input field* is a field suitable for the F format code that is assumed to have no fractional digits, unless a decimal point appears within the field.

When the corresponding list item is of COMPLEX type, the input form consists of a left parenthesis, an ordered pair of numeric input fields separated by a comma, and a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of logical type, the input form must not include either slashes or commas among the optional characters permitted for the L format code.

When the corresponding list item is of character type, the input form consists of a nonempty string of characters enclosed in apostrophes. Each apostrophe within a character constant must be represented by two consecutive apostrophes without an intervening blank or the end of the record. Character constants may be continued from the end of one record to the beginning of the next record. The end of the record does not cause a blank or any other character to become part of the constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

For example, let *len* be the length of the list item, and let *w* be the length of the character constant. If *len* is less than or equal to *w*, the leftmost *len* characters of the constant are transmitted to the list item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the list item and the remaining *len-w* characters of the list item are filled with blanks. The effect is that the constant is assigned to the list item in a character assignment statement.

A null value is specified by having no characters between successive separators, by having no characters preceding the first value separator in the first record read by each execution of a list-directed input statement, or by the *r** form. A null value has no effect on the definition status by the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value may not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant. The end of a record following any other separator, with or without separating blanks, does not specify a null value.

A slash encountered as a value separator during execution of a list-directed input statement causes termination of execution of that input statement after the assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them.

All blanks in a list-directed input record are considered part of some value separator, except for the following:

- Blanks embedded in a character constant
- Embedded blanks surrounding the real or imaginary part of a complex constant
- Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma

Output: Except as noted, the form of the values produced is the same as that required for input. With the exception of character constants, the values are separated by one of the following:

- One or more blanks
- A comma, optionally preceded by one or more blanks and optionally followed by one or more blanks

VS FORTRAN Version 2 may begin new records as necessary but, except for complex constants and character constants, the end of a record must not occur within a constant, and blanks must not appear within a constant.

Logical output constants are T for the value .TRUE. and F for the value .FALSE..

Integer output constants are produced with the effect of an I*w* edit descriptor for some reasonable value of *w*.

Real and double precision constants are produced with the effect of either an F format code or an E format code, depending on the magnitude *x* of the value and a range:

```
10**d1 ≤ x < 10**d2
```

where *d1* and *d2* are processor-dependent integer values. If the magnitude *x* is within this range, the constant is produced using 0PF*w.d*; otherwise, 1PE*w.d*E*e* is used. Reasonable processor-dependent values are used for each of the cases involved.

Complex constants are enclosed in parentheses, with a comma separating the real and imaginary parts. The end of a record may occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.

Character constants produced:

- Are not delimited by apostrophes
- Are not preceded or followed by a value separator
- Have each internal apostrophe represented externally by one apostrophe
- Have a blank character inserted at the beginning of any record that begins with the continuation of a character constant from the preceding record

If two or more successive values in an output record produced have identical values, the sequence of identical values is written.

Slashes, as value separators, and null values are not produced by list-directed formatting.

Each output record begins with a blank character to provide carrier control if the record is printed.

FUNCTION Statement

The FUNCTION statement identifies a function subprogram consisting of a FUNCTION statement followed by other statements that may include one or more RETURN statements. It is an independently written program that is executed wherever its name is referred to in another program.

Syntax

[*type*] **FUNCTION** *name* ([*arg1* [, *arg2*...]])

type

is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER[**len1*]

where:

**len1*

is the length specification. It is optional; if omitted, it is assumed to be 1. It may be an unsigned, nonzero, integer constant, an integer constant expression enclosed in parentheses, or an asterisk enclosed in parentheses. The expression can only contain integer constants; it must not include names of integer constants.

If the name is of character type, all entry names must be of character type, and lengths must be the same. If one length is specified as an asterisk, all lengths must be specified as asterisks.

name

is the name of the function.

*name*len2*

is the name of the function.

where:

**len2*

is a positive, nonzero, unsigned integer constant. It represents one of the permissible length specifications for its associated type. (See "Data Types and Lengths" on page 22.) It may be included only when *type* is specified as INTEGER, REAL, COMPLEX, or LOGICAL. It must not be used when DOUBLE PRECISION or CHARACTER is specified.

arg

is a dummy argument. It must be a variable or array name that may appear only once within the FUNCTION statement or dummy procedure name. If there is no argument, the parentheses must be present. (See "Dummy Arguments in a Function Subprogram" on page 123.)

A type declaration for a function name may be made by the predefined convention, by an IMPLICIT statement, by an explicit specification in the FUNCTION statement, or by an explicit type specification statement within the function subprogram. If the type of a function is specified in a FUNCTION statement, the function name must not appear in an explicit type specification statement.

The name of a function must not be in any other nonexecutable statement except a type statement.

Because the FUNCTION statement is a separate program unit, there is no conflict if the variable names and statement labels within it are the same as those in other program units.

The FUNCTION statement must be the first statement in the subprogram. The function subprogram may contain any FORTRAN statement except a SUBROUTINE statement, another FUNCTION statement, a BLOCK DATA statement, or a PROGRAM statement. If an IMPLICIT statement is used in a function subprogram, it must follow the FUNCTION statement and may only be preceded by another IMPLICIT statement, or by a PARAMETER, FORMAT, or ENTRY statement.

The name of the function (or one of the ENTRY names) must appear as a variable name in the function subprogram and must be assigned a value at least once during the execution of the subprogram in one of the following ways:

- As the variable name to the left of the equal sign in an arithmetic, logical, or character assignment statement
- As an argument of a CALL statement that will cause a value to be assigned in the subroutine referred to
- In the list of a READ statement within the subprogram
- As one of the parameters in an INQUIRE statement that is assigned a value within the subprogram
- As a DO- or implied DO-variable
- As the result of the IOSTAT specification in an I/O statement

The value of the function is the last value assigned to the name of the function when a RETURN or END statement is executed in the subprogram. For additional information on RETURN and END statements in a function subprogram, see "RETURN Statement" on page 196 and "END Statement" on page 83.

The function subprogram may also use one or more of its arguments to return values to the calling program. An argument so used must appear:

- On the left side of an arithmetic, logical, or character assignment statement

- In the list of a READ statement within the subprogram
- As an argument in a function reference that is assigned a value by the function referred to
- As an argument in a CALL statement that is assigned a value in the subroutine referred to
- As one of the parameters in an INQUIRE statement

The dummy arguments of the function subprogram (for example, *arg1*, *arg2*, *arg3*, ...) are replaced at the time of invocation by the actual arguments supplied in the function reference in the calling program.

If a function dummy argument is used as an adjustable array name, the array name and all the variables in the array declarators (except those in the common block) must be in the dummy argument list. See "Size and Type Declaration of an Array" on page 26.

If the predefined convention is not correct, the function name must be typed in the program units that refer to it. The type and length specifications of the function name in the function reference must be the same as those of the function name in the FUNCTION statement.

Except in a character assignment statement, the name of a character function whose length specification is an asterisk must not be the operand of a concatenation operation.

The length specified for a character function in the program unit that refers to the function must agree with the length specified in the subprogram that specifies the function. There is always agreement of length if the asterisk is used in the referenced subprogram to specify the length of the function.

Actual Arguments in a Function Subprogram

The actual arguments in a function reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced function. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type.

If an actual argument is of type character, the associated dummy argument must be of type character and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument is greater than the length of an associated dummy argument, the leftmost characters of the actual argument are associated with the dummy argument.

An actual argument in a function reference must be one of the following:

- An array name
- An intrinsic function name
- An external procedure name
- A dummy argument name

- An expression, except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses (unless the operand is the name of a constant).

For an entry point in a function subprogram, see "ENTRY Statement" on page 86.

Dummy Arguments in a Function Subprogram

The dummy arguments of a function subprogram appear after the function name and are enclosed in parentheses. They are replaced at the time of invocation by the actual arguments supplied in the function reference.

Dummy arguments must adhere to the following rules:

- None of the dummy argument names may appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, SAVE, INTRINSIC, or NAMELIST statement, except as NAMELIST or common block names, in which case the names are not associated with the dummy argument names.
- A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, ENTRY, or statement function definition in the same program unit.
- The dummy arguments must correspond in number, order, and type to the actual arguments.
- If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, an array element, a substring, or an array. A constant, name of constant, subprogram name, or expression should not be written as an actual argument unless the programmer is certain that the corresponding dummy argument has not been assigned a value in the subprogram.
- A referenced subprogram cannot assign new values to dummy arguments that are associated with other dummy arguments within the subprogram or with variables in the common block.

Valid FUNCTION Statements:

1. Definition of function subprogram SUFFIX:

   ```
   CHARACTER*10 FUNCTION SUFFIX(STR)
   CHARACTER*7 STR
   SUFFIX = STR // 'SUF'
   END
   ```

   Use of function subprogram SUFFIX:

   ```
   CHARACTER*10 NAME, SUFFIX
   ...
   NAME = SUFFIX(NAME(1:7))
   ```

2. Definition of function subprogram CUBE. This illustrates a function defined without dummy arguments:

   ```
   REAL FUNCTION CUBE*16()
   COMMON /COM1/ A
   CUBE = A * A * A
   END
   ```

   Use of function subprogram CUBE. Functions defined without dummy arguments must be invoked with the null parentheses.

```
REAL*16 A,X
COMMON /COM1/ A
A = 1.6

X = CUBE()
```

3. Function IADD illustrates assigning a value to the function name (in this case, IADD) by means of an argument of a CALL statement.

   ```
   FUNCTION IADD( M )
   ...
   CALL SUBA (IADD, M)
   RETURN
   END
   ```

   Definition of subroutine SUBA:

   ```
   SUBROUTINE SUBA (J,K)
   J = 10 + K
   RETURN
   END
   ```

4. Function IREAD illustrates assigning a value to the name of a function (in this case, IREAD) by means of an I/O list of a READ statement within the function definition.

   ```
   FUNCTION IREAD ()
   READ *, IREAD
   RETURN
   END
   ```

5. Function SUM illustrates the use of adjustable dimensions.

   ```
      INTEGER FUNCTION SUM(ARRY, M, N)
      INTEGER M, N, ARRY(M, N)

      SUM = 0
      DO 10 I = 1, M
      DO 10 J = 1, N
   10 SUM = SUM + ARRY(I,J)
      RETURN
      END
   ```

   Use of function subprogram SUM:

   ```
   DIMENSION IARRAY(20,30)
   INTEGER SUM
   ...
   IVAR = SUM(IARRAY, 20, 30)
   ```

Invalid FUNCTION Statements:

Assume the following function definition:

```
REAL FUNCTION BAD(ARG)

IF ( ARG .EQ. 0.0 ) ARG = 1.0
BAD = 123.4/ARG

RETURN
END
```

The following use of BAD is illegal, because the actual argument is an expression, and BAD may assign a value to its dummy argument.

```
     X = BAD( 6.0 * X )
```

The following use of BAD is also illegal, because the actual argument is a constant.

```
     X = BAD( 12.3 )
```

GO TO Statements

GO TO statements transfer control to an executable statement in the program unit. There are three GO TO statements:

- Assigned GO TO
- Computed GO TO
- Unconditional GO TO

Assigned GO TO Statement

The assigned GO TO statement transfers control to the statement labeled *stl1*, *stl2*, *stl3*, ... depending on whether the current assignment of *i* is *stl1*, *stl2*, *stl3* ... respectively. (See "ASSIGN Statement" on page 51.)

Syntax

GO TO *i* [[,] (*stl1* [,*stl2*] [,*stl3*] ...)]

i

is an integer variable (not an array element) of length 4 that has been assigned a statement label by an ASSIGN statement.

stl

is the statement label of an executable statement in the same program unit as the assigned GO TO statement.

The list of statement labels, that is, (*stl*1, *stl*2, *stl*3 ...), is optional. If omitted, the preceding comma must be omitted. If the list of statement labels is specified, the preceding comma is optional. The statement label assigned to *i* must be one of the statement labels in the list. The statement label may appear more than once in the list.

The ASSIGN statement that assigns the statement label to *i* must appear in the same program unit as the assigned GO TO statement that is using this statement label.

For example, in the statement:

```
GO TO N, (10, 25, 8)
```

If the current assignment of the integer variable N is statement number 8, the statement labeled 8 is executed next. If the current assignment of N is statement number 10, the statement labeled 10 is executed next. If N is assigned statement label 25, statement 25 is executed next.

At the time of execution of an assigned GO TO statement, the current value of *i* must have been assigned the statement label of an executable statement (not a FORMAT statement) by the previous execution of an ASSIGN statement.

If, at the time of the execution of an assigned GO TO statement, the current value of *i* contains an integer value, assigned directly or through EQUIVALENCE, COMMON, or argument passing, the result of the GO TO is unpredictable. If the integer variable *i* is a dummy argument in a subprogram, it must be assigned a statement label in the subprogram, and also used in an assigned GO TO in that subprogram. An integer variable used as an actual argument in a subprogram reference may not be used in an assigned GO TO in the invoked subprogram unless it is redefined in the subprogram.

Any executable statement immediately following the assigned GO TO statement should have a statement label; otherwise, it can never be referred to or executed. An assigned GO TO statement cannot terminate the range of a DO.

Example:

```
ASSIGN 150 TO IASIGN
IVAR=150.
GO TO IASIGN
```

Computed GO TO Statement

The computed GO TO statement transfers control to the statement labeled *stl1*, *stl2*, or *stl3*, ... depending on whether the current value of *m* is 1, 2, or 3, ... respectively.

Syntax

GO TO (*stl1* **[,** *stl2***] [,** *stl3***], ...) [,]** *m*

stl

is the statement label of an executable statement in the same program unit as the computed GO TO statement. The same label may appear more than once within the parentheses.

m

is an integer expression. The comma before *m* is optional. If the value of *m* is outside the range $1 \le m \le n$, where n is the number of statement labels, the next statement is executed.

A computed GO TO statement may terminate the range of a DO.

Example:

```
171  GO TO(172,173,174,173) INT(A)
172  A = A + 1.0
     GO TO 174
173  A = A + 1.0
174  CONTINUE
```

Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the statement specified by the statement label. Every subsequent execution of this GO TO statement results in a transfer to that same statement.

Syntax

GO TO *stl*

stl

is the statement label of an executable statement in the same program unit as the unconditional GO TO statement.

Any executable statement immediately following this statement must have a statement label; otherwise, it can never be referred to or executed.

An unconditional GO TO cannot terminate the range of a DO-loop.

Example:

```
    GO TO 5
999 I = I + 200
     .
     .
     .
  5 I = I + 1
```

IF Statements

The IF statements specify alternative paths of execution, depending on the condition given. There are three forms of the IF statement:

- Arithmetic IF
- Block IF

  ELSE
  ELSE IF
  END IF
- Logical IF

Arithmetic IF Statement

The arithmetic IF statement transfers control to the statement labeled *stl1*, *stl2*, or *stl3* when the value of the arithmetic expression (*m*) is less than, equal to, or greater than zero, respectively. The same statement label may appear more than once within the same IF statement.

Syntax

IF (*m*) *stl1*, *stl2*, *stl3*

m

is an arithmetic expression of any type except complex.

stl

is the label of an executable statement in the same program unit as the IF statement.

An arithmetic IF statement cannot terminate the range of a DO-loop.

Any executable statement immediately following this statement must have a statement label; otherwise, it can never be referred to or executed.

Block IF Statement

The block IF statement is used with the END IF statement and, optionally, the ELSE IF and ELSE statements to control the execution sequence.

Syntax

IF (*m*) THEN

m

is any logical expression.

Two terms are used in connection with the block IF statement: **IF-level** and **IF-block**.

IF-level The number of *IF-levels* in a program unit is determined by the number of *sets* of block IF statements (IF (*m*) THEN and END IF statements).

The *IF-level* of a particular statement (*stl*) is determined with the formula:

```
n1 - n2
```

where:

n1

is the number of block IF statements from the beginning of the program unit up to and including the statement (*stl*).

n2

is the number of END IF statements in the program unit up to, but not including, the statement (*stl*).

IF-block An *IF-block* begins with the first statement after the block IF statement (IF (*m*) THEN), ends with the statement preceding the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement, and includes all the executable statements in between. An IF-block is empty if there are no executable statements in it.

Transfer of control into an IF-block from outside the IF-block is prohibited.

Execution of a block IF statement evaluates the expression *m*. If the value of *m* is true, normal execution sequence continues with the first statement of the IF-block, which is immediately following the IF (*m*) THEN. If the value of *m* is true, and the IF-block is empty, control is transferred to the next END IF statement that has the same IF-level as the block IF statement. If the value of *m* is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement.

If the execution of the last statement in the IF-block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the block IF statement that precedes the IF-block.

A block IF statement cannot terminate the range of a DO.

END IF Statement

The END IF statement concludes an IF-block. Normal execution sequence continues.

Syntax

END IF

For each block IF statement, there must be a matching END IF statement in the same program unit. A matching END IF statement is the next END IF statement that has the same IF-level as the block IF statement.

An END IF statement cannot terminate the range of a DO.

Valid END IF Statements:

The following is the general form of a single alternative block IF statement (in other words, no ELSE or ELSE IF statements are in the IF-block).

```
      IF ( m ) THEN
C
C        EXECUTION SEQUENCE WHEN THE VALUE OF m IS TRUE
C
      ...
      ENDIF
C
C        IF m IS FALSE, EXECUTION CONTINUES HERE
C
      ...
```

The following is an example of a single alternative IF.

```
      IF ( INDEX .EQ. 0) THEN
         PRINT *, 'KEY NOT FOUND'
         INDEX = - 1
      ENDIF
      ...
```

ELSE Statement

The ELSE statement is executed if the preceding block IF or ELSE IF condition is evaluated as FALSE. Normal execution sequence continues.

Syntax

ELSE

An ELSE-block consists of all the executable statements after the ELSE statement up to, but not including, the next END IF statement that has the same IF-level as the ELSE statement. An ELSE-block may be empty.

Within an IF-block, you can have only one ELSE.

Transfer of control into an ELSE-block from outside the ELSE-block is prohibited. The statement label, if any, of an ELSE statement must not be referred to by any statement (except an AT statement of a DEBUG packet). An ELSE statement cannot terminate the range of a DO.

Valid ELSE Statements:

The following is the general form of the double alternative block IF statement (in other words, IF-block contains an ELSE statement but no ELSE IF statements).

```
      IF (m) THEN
C
C        EXECUTION SEQUENCE WHEN THE VALUE OF m IS TRUE
C

⋮
      ELSE
C
C        EXECUTION SEQUENCE WHEN THE VALUE OF m IS FALSE
C

⋮
      ENDIF
```

The following is an example of a double alternative block IF.

```
      IF( X .GE. Y ) THEN
         LARGE = X
      ELSE
         LARGE = Y
      ENDIF
```

ELSE IF Statement

The ELSE IF statement is executed if the preceding block IF condition is evaluated as false.

Syntax

ELSE IF (*m*) THEN

m
is any logical expression.

An ELSE IF block consists of all the executable statements after the ELSE IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement. An ELSE IF block may be empty.

If the value of the logical expression *m* is true, normal execution sequence continues with the first statement of the ELSE IF block.

If the value of *m* is true and the ELSE IF block is empty, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement.

If the value of *m* is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement.

Transfer of control into an ELSE IF block from outside the ELSE IF block is prohibited. The statement label (*stl*), if any, of the ELSE IF statement must not be referred to by any statement (except an AT statement of a DEBUG packet).

If execution of the last statement in the ELSE IF block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement that precedes the ELSE IF block.

An IF-THEN-ELSE structure can contain a maximum of 125 nested ELSE IF blocks. An ELSE IF statement cannot terminate the range of a DO.

Valid ELSE IF Statements:

The following are the general forms of the multiple alternative block-IF statement.

```
IF ( m ) THEN
```

1. Execution sequence when the value of *m* is true.

   ```
   ...
   ELSE IF ( ml ) THEN
   ```

2. Execution sequence when the value of *m* is false and the value of *m1* is true.

   ```
   ...
   ELSE
   ```

3. Execution sequence when the values of both *m* and *m1* are false.

   ```
   ...
   ENDIF
   ```

The following is the second form of the multiple alternative block-IF.

```
IF ( m ) THEN
```

1. Execution sequence when the value of *m* is true.

   ```
   ...
   ELSE IF ( ml ) THEN
   ```

2. Execution sequence when the value of *m* is false and the value *m1* is true.

   ```
   ...
   ENDIF
   ```

3. Execution continues here, following execution of the block-IF.

   ```
   ...
   ```

The following is an example of multiple alternative block-IF.

```
CHARACTER*5 C

IF ( C .EQ. 'RED  ' ) THEN
  PRINT *, ' COLOR IS RED'
ELSEIF ( C .EQ. 'BLUE ' ) THEN
  PRINT *, ' COLOR IS BLUE'
ELSEIF ( C .EQ. 'WHITE') THEN
  PRINT *, ' COLOR IS WHITE'
ELSE
  PRINT *, ' COLOR IS NOT SET'
  C = 'PLAID'
  PRINT *, ' COLOR IS NOW PLAID'
ENDIF
```

Logical IF Statement

The logical IF statement evaluates a logical expression and executes or skips a statement, depending on whether the value of the expression is true or false, respectively.

Syntax

IF (*m*) *st*

m

is any logical expression.

st

is any executable statement except:

- A DO statement
- Another logical IF statement
- An END statement
- A block IF, ELSE IF, ELSE, or END IF statement.
- A TRACE ON or TRACE OFF statement
- An INCLUDE statement
- A DISPLAY statement

The statement *st* must not have a statement label.

The execution of a function reference in *m* is permitted to effect entities in the statement *st*.

The logical IF statement containing *st* may have a statement label. If a logical IF statement terminates the end of a DO loop, it may not contain a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

Example:

```
   IF(A.LE.0.0) GO TO 25
   C = D + E
   IF (A.EQ.B) ANSWER = 2.0*A/C
   F = G/H
25 W = X**Z
```

⋮

IMPLICIT Statement

The IMPLICIT statement can be used to confirm or change the default implied types or it may be used to void implied typing altogether.

Syntax

IMPLICIT *type* (*a* **[**, *a* **]** ...) **[**, *type* (*a* **[**, *a* **]** ...) **]** ...

IMPLICIT NONE

type

is CHARACTER[**len*], COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, REAL, COMPLEX*8, COMPLEX*16, COMPLEX*32, INTEGER*2, INTEGER*4, LOGICAL*1, LOGICAL*4, REAL*4, REAL*8, or REAL*16.

len

specifies the length of a character entity. It is either an unsigned, nonzero, integer constant, or a positive constant expression enclosed in parentheses that has a positive value.

a

is a single letter or range of letters. A range of letters is denoted by a_1-a_2, where a_1 precedes a_2 alphabetically. (The currency symbol, $, is considered to follow the letter Z.) A range of letters has the same effect as specifying each letter within the range separately. For example, IMPLICIT INTEGER (A-C) is equivalent to IMPLICIT INTEGER (A,B,C).

An IMPLICIT NONE statement voids all implied typing. A TYPE statement must then be used to specify explicitly the data type of a name. The IMPLICIT NONE statement must be the only IMPLICIT statement in a program unit. A PARAMETER statement may not precede an IMPLICIT NONE statement.

The IMPLICIT statement specifies the implied type of any name that begins with any letter in the specification. The IMPLICIT statement does not change the implied type of names which begin with non-EBCDIC double-byte characters. A TYPE statement may be used to confirm or change the implied type of a name.

For any name, type specification by an IMPLICIT statement may be overridden or confirmed by the appearance of that name in an explicit type specification statement.

An IMPLICIT statement has no effect on names of intrinsic functions.

Valid IMPLICIT Statements:

```
IMPLICIT INTEGER(A,B,G-H), REAL(I-K), LOGICAL(L,M,N)

IMPLICIT COMPLEX(C-F)

IMPLICIT INTEGER(W-$)
```

For the following two IMPLICIT statements:

```
IMPLICIT DOUBLE PRECISION (A-C, F)
IMPLICIT LOGICAL (E,L), CHARACTER (D,G,H)
```

FORTRAN will treat the implicit type of names as follows:

Names beginning with:	Have data type:	Have length:
A,B,C,F,a,b,c,f	DOUBLE PRECISION	8
E,L,e,l	LOGICAL	4
D,G,H,d,g,h	CHARACTER	1
I-K,M,N,i-k,m,n	INTEGER	4 (defaults)
O-Z,o-z,$	REAL	4 (defaults)

INCLUDE Statement

INCLUDE is a compiler directive. It inserts a specified statement or a group of statements into a program unit.

A facility called conditional INCLUDE provides a means for selectively activating INCLUDE statements within the source program during compilation. The included files are specified by means of the CI compiler option. For more information about the CI compiler option and how to use the INCLUDE directive, see *VS FORTRAN Version 2 Programming Guide*.

There are two forms of the INCLUDE statement syntax. The first form, which requires a file definition (FILEDEF command, ALLOCATE command, or DD statement), allows you to specify the name of the source library member which is to be included. The second form lets you refer directly to CMS files or MVS data sets without the need for a file definition.

Syntax 1

INCLUDE (*member-name***) [***n***]**

member-name

a sequence of 1 to 8 letters or digits, the first of which must be a letter. *member-name* is the name of a member in the source library which is to be included.

n

is the value used to decide whether to include the file during compilation. When *n* is not specified, the file is always included. When *n* is specified, the file is included only if the number appears in the list of identification numbers on the CI compiler option. The range of *n* is 1 to 255.

Syntax 2

INCLUDE *char-constant*

char-constant

is a character constant whose value, when trailing blanks are removed, is a system-dependent file specifier naming the file to be included. This type of INCLUDE statement conforms to the Systems Application Architecture. The file specifier is in one of the following formats:

- **On VM:** *filename* [*filetype* [*filemode*]] [(*member*)]

  filename filetype
  : each is a sequence of 1 to 8 characters chosen from letters, numbers, and special characters (hyphen (-), colon (:), underscore (_), @, #, and +). If *filetype* is not specified, a file type of FORTRAN is assumed.

  filemode
  : from 1 to 2 characters whose first character ranges from A to Z and whose second character ranges from 0 to 6. If the second character is not specified, 1 will be assumed. If *filemode* is omitted, A1 will be used. An asterisk (*) can also be specified for the file mode. If an asterisk is specified, the CMS search order is used to locate the first file with the same file name and file type.

  member
  : a member name, 1 to 8 characters long, within a MACLIB. When *member* is specified, the file type must be MACLIB. If no file type is specified, MACLIB is assumed.

  Note: Lower case letters are equivalent to upper case letters even though it is possible to create file names with lowercase in CMS. One or more blanks must separate each part of the CMS file identifier.

- **On MVS:** *dsn* [(*member*)]

  dsn
  : a fully qualified MVS data set name. It consists of qualifiers, each of which are 1 through 8 characters long, separated by periods to a maximum length of 44 characters. The character set is restricted to letters, numbers and special characters (# and @). The first character must be a letter, #, or @.

  member
  : a member within a partitioned data set. Partitioned data set names must be no longer than 54 characters (44 for the data set name and 8 for the member name plus the parentheses around the member name). Partitioned data sets are valid only for non-VSAM files.

  Note: Lower case letters are equivalent to upper case letters.

The following rules apply to INCLUDE:

- INCLUDE must not be continued.
- The group of statements inserted by the INCLUDE may contain any FORTRAN source statements, including other INCLUDE statements.
- The first non-comment line of the included group of statements must not be a continuation line.
- An INCLUDE of a group must not contain an INCLUDE statement that refers to a currently open INCLUDE group (that is, recursion is not permitted).
- Multiple INCLUDE statements may appear in the original source program.
- INCLUDE statements may appear anywhere in a source program before the END statement, except as the trailer of a logical IF statement. An END statement may be part of the included group.

- The statements in the group being included must be in the same form as the source program being compiled; that is, either fixed form or free form.
- After the inclusion of all groups, the resulting program must follow all rules for sequencing of statements.
- An included file may not contain an @PROCESS statement.

Valid INCLUDE Statements:

For Syntax 1

```
INCLUDE (MYFILE)

INCLUDE (DATA) 2
```

For Syntax 2

VM Examples

```
INCLUDE 'CONSTANT'

INCLUDE 'COMMON PROJ_01'

INCLUDE 'MASKS-1 INCLUDE Z1'

INCLUDE 'OLDPROJ MACLIB (CONTROL)'
```

MVS Examples

```
INCLUDE 'MORRIS.HISTO.PACKAGE'

INCLUDE 'TJWATSON.PROJ1.FORT.INCL(COMMON)'

INCLUDE 'CHOPIN.PROF2.FORT.CONST(MASKS)'
```

INQUIRE Statement

The INQUIRE statement supplies information about a particular file or about a particular external unit. By using INQUIRE, you can check a file's existence and its connection to an external unit. File existence can be determined regardless of whether the file is connected to a unit; the values returned by the INQUIRE statement reflect the status of the file and file connection at the time the statement executes. There are three types of INQUIRE statements:

1. INQUIRE by file
2. INQUIRE by unit
3. INQUIRE by unnamed file

INQUIRE by File

Allows you to determine the file existence and connection status and other properties of a named file. For a discussion of named and unnamed files, see "Input/Output Semantics" on page 46.

Syntax

INQUIRE
  (FILE=*fn*
  [,*specifier-list***])**

FILE=*fn*

fn is the name of the file and *must* be preceded by FILE=. It may take one of the following formats:

- *ddname*

  a character expression whose value, when any trailing blanks are removed, is the name by which VS FORTRAN identifies the file definition. The expression must be 1 to 8 characters, the first one being a letter, #, or @, and the other seven being letters or digits, including # and @.

- **On VM:** */filename filetype* [*filemode*] [(*member*)]

  a character expression whose value, when any trailing blanks are removed, is the file name, file type, and file mode on VM.

  The slash (/) indicates that the file identifier is not a ddname; therefore, it must appear as the first character of the expression if this format is used.

  filename filetype
  each is a sequence of 1 to 8 characters chosen from letters, numbers and special characters (hyphen (-), colon (:), underscore (_), @, #, +).

  filemode
  from 1 to 2 characters long whose first character ranges from A to Z and whose second character ranges from 0 to 6.

  - If the second character is not specified, 1 will be assumed.
  - If *filemode* is omitted, A will be used as the file mode.
  - If an asterisk (*) is specified for the file mode, the standard CMS search order will be used; if the file is found and the NAME specifier is used, then INQUIRE will return the actual file mode.

  member
  a member name, 1 to 8 characters long, and the parentheses are required to indicate that it is a member name. *member* may be used only when referring to a LOADLIB, MACLIB, or TXTLIB filetype.

  Note: Lowercase letters are equivalent to uppercase letters. One or more blanks must separate each part of the CMS file identifier.

  If this form is used, the FILEINF service subroutine may be used to set up the file characteristics prior to issuing the INQUIRE statement. If FILEINF is not used, the unit attribute table values will be used instead. For more information, see "FILEINF Subroutine" on page 283.

- **On MVS:** */dsn* [(*member*)]

  a character expression whose value, when any trailing blanks are removed, is the data set name on MVS.

  The slash (/) is used to indicate that the file identifier is not a ddname; therefore, it must appear as the first character of the expression if this form is used.

  dsn
  a fully qualified MVS data set name. It consists of qualifiers, each of which are 1 to 8 characters long, separated by periods, to a

maximum length of 44 characters. The character set is restricted to letters, numbers and special characters (# and @). The first character must be a letter, # or @.

member
a member within a partitioned data set. Partitioned data set names must be no longer than 54 characters (44 for the data set name and 8 for the member name plus the parentheses around the member name). Partitioned data sets are valid only for non-VSAM files.

Note: Trailing blanks are ignored and lower case is equivalent to upper case.

If this form is used, the FILEINF service routine may be used to set up the file characteristics prior to issuing the INQUIRE statement. If FILEINF is not used, the unit attribute table values will be used instead. For more information, see "FILEINF Subroutine" on page 283.

INQUIRE by Unit

Allows you to determine the existence of a unit, whether the unit is connected to a file, and, if the unit is connected, what the properties are of the unit and file connection. For a general discussion of file and unit connection, see "Input/Output Semantics" on page 46.

Syntax

INQUIRE
([UNIT=]*un*
[,*specifier-list***])**

UNIT=*un*
un is an integer expression of length 4 whose value is the external unit identifier. *un* can be optionally preceded by UNIT=. If UNIT= is not specified, *un* must appear immediately following the left parenthesis.

INQUIRE by Unnamed File

Allows you to determine the file existence and connection status for an unnamed file, as well as other properties of the file. For a discussion of named and unnamed files, see "Input/Output Semantics" on page 46.

Syntax 1

INQUIRE
([UNIT=]*un***,**
FILE=*fn*
[,*specifier-list***])**

Syntax 2

INQUIRE
(FILE=*fn*
[,*specifier-list***])**

For Syntax 1:

UNIT=*un*

un is an integer expression of length 4 whose value is the external unit identifier.

FILE=*fn*

fn is an 8-byte character expression whose value is blanks. *fn* may be specified as a character constant, as shown in the following example:

```
INQUIRE (UNIT=un,FILE=' ', ...)
```

If Syntax 1 of INQUIRE by unnamed file is used, information is requested about an unnamed file that is currently connected to the unit referred to by *un*. (The unnamed file being inquired about must have been connected by an ENDFILE, OPEN, READ, or WRITE statement.) If the unit referred to by *un* exists but there is no unnamed file currently connected to it, VS FORTRAN attempts to determine file existence as follows:

1. Checks for a file definition for: FTnnK01 (where nn is the two-digit unit identifier).
2. Checks for a file definition for: FTnnF001 (where nn is the two-digit unit identifier).

If a file definition is found for FTnnF001, VS FORTRAN will return the file existence properties for that file; otherwise, VS FORTRAN will continue to determine the existence of FTnnF001.

For Syntax 2:

FILE=*fn*

fn must be a character expression whose value, when trailing blanks are removed, is one of the default ddnames used for unnamed files. These default ddnames take one of the following forms.

- FTnnFmmm, where nn is the two-digit unit identifier and mmm is the three-digit sequence number
- FTnnKkk, where nn is the two-digit unit identifier and kk is the two-digit sequence number for a keyed file
- FTERRsss, where sss is the three-digit MTF subtask number
- FTPRTsss, where sss is the three-digit MTF subtask number

Optional Specifiers

The optional specifiers are listed in Figure 23 on page 140 and may be specified in any order. Each specifier cannot appear more than once in an INQUIRE statement. The same variable or array element cannot be designated for more than one specifier in the same INQUIRE statement.

Note: Except for the IOSTAT, ERR, and PASSWORD specifiers, the variables or array elements given in the optional specifiers become defined under different conditions depending on the form of INQUIRE you are using. See Figure 24 on page 145 for details.

```
ERR=stl
IOSTAT=ios
EXIST=exs
OPENED=opn
NAMED=nmd
NAME=nam
SEQUENTIAL=seq
DIRECT=dir
KEYED=kyd
FORMATTED=fmt
UNFORMATTED=unf
NUMBER=num
ACCESS=acc
FORM=frm
CHAR=chr
RECL=rcl
NEXTREC=nxr
BLANK=blk
ACTION=act
WRITE=wri
READ=ron
READWRITE=rwr
PASSWORD=pwd
KEYID=kid
KEYLENGTH=kle
KEYSTART=kst
KEYEND=ken
LASTRECL=lrl
LASTKEY=lky
```

Figure 23. Optional Specifiers on the INQUIRE Statement

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the INQUIRE statement. If an error occurs, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or integer array element of length 4. *ios* is set to a positive value if an error is detected; it is set to zero is no error is detected. VSAM return codes and reason codes are placed in *ios*.

EXIST=*exs*

exs is a logical variable or logical array element of length 4.

If you are using INQUIRE by file: *exs* is assigned the value true if the file of the specified name exists; otherwise, it is assigned the value false. *exs* becomes undefined if an error occurs.

If you are using INQUIRE by unit: *exs* is assigned the value true if the specified unit exists; otherwise it is assigned the value false.

If you are using INQUIRE by unnamed file: *exs* is assigned the value true if the unit exists and the unnamed file both exist; otherwise, *exs* is assigned the value false. *exs* becomes undefined if an error occurs.

OPENED=*opn*

opn is a logical variable or logical array element of length 4. *opn* becomes undefined if an error occurs.

If you are using INQUIRE by file: *opn* is assigned the value true if the file specified is connected to a unit; otherwise, it is assigned the value false.

If you are using INQUIRE by unit: *opn* is assigned the value true if the unit specified is connected to a file; otherwise, it is assigned the value false.

Note: For INQUIRE by unit on VM, *opn* is always assigned the value true, regardless of whether the unit is connected to a file or not.

If you are using INQUIRE by unnamed file: *opn* is assigned the value true if the unnamed file referred to by the FILE specifier is connected to a corresponding unit; otherwise, it is assigned the value false.

NAMED=*nmd*

nmd is a logical variable or logical array element of length 4.

If you are using INQUIRE by file: *nmd* is assigned the value true.

If you are using INQUIRE by unit: If the file connected to the unit is a named file, then *nmd* is assigned the value true.

If you are using INQUIRE by unnamed file: *nmd* is always assigned the value false.

NAME=*nam*

nam is a character variable or character array element.

If you are using INQUIRE by file: *nam* is assigned the value of either the ddname, the CMS file identifier, or the MVS data set name, depending on what name was used when the file was connected. The system dependent names are returned in the following formats:

Under CMS:

'/*filename filetype filemode*' or '/*filename filetype filemode* (*member*)'

where *filename*, *filetype*, and *member* are 8 characters long and *filemode* is two characters long.

Under MVS:

'/*dsn*' or '/*dsn* (*member*)'

where *dsn* is 44 characters long and *member* is 8 characters long.

Note: If *filename*, *filetype*, *filemode*, *member* or *dsn* are shorter than the lengths stated above, they will be padded with blanks. If *member* is returned, there will be one blank before the left parenthesis. The variable used to hold the name returned by INQUIRE (*nam*) must be declared to be at least as long as 21 characters for CMS files (32 for CMS files that include *member*) and at least as long as 45 characters for MVS files (56 for MVS files that include *member*).

If you are using INQUIRE by unit: If the file connected to the designated unit has a name, then *nam* is assigned the value of that name—either the ddname, the CMS file identifier or the MVS data set name, depending on what name was used when the file was connected, according to the formats described for INQUIRE by file above. Otherwise, *nam* becomes undefined.

If you are using INQUIRE by unnamed file: *nam* becomes undefined.

SEQUENTIAL=*seq*

seq is a character variable or character array element. It is assigned a value of YES if the file can be connected for sequential access, NO if the file cannot, and UNKNOWN if it is not possible to determine whether the file can be connected for sequential access.

DIRECT=*dir*

dir is a character variable or a character array element. It is assigned a value of YES if the file can be connected for direct access, NO if the file cannot, and UNKNOWN if it is not possible to determine whether the file can be connected for direct access.

KEYED=*kyd*

kyd is a character variable or a character array element. It is assigned a value of YES if the file can be connected for keyed access, NO if the file cannot, and UNKNOWN if it is not possible to determine whether the file can be connected for keyed access.

FORMATTED=*fmt*

fmt is a character variable or a character array element. It is assigned a value of YES if the file can be connected for formatted access, NO if the file cannot, and UNKNOWN if it is not possible to determine whether the file can be connected for formatted access.

UNFORMATTED=*unf*

unf is a character variable or a character array element. It is assigned a value of YES if the file can be connected for unformatted access, NO if the file cannot, and UNKNOWN if it is not possible to determine whether the file can be connected for unformatted access.

NUMBER=*num*

num is an integer variable or integer array element of length 4. *num* is assigned the value of the unit identifier for the unit to which the file is connected.

ACCESS=*acc*

acc is a character variable or character array element. *acc* is assigned a value that corresponds to the type of file connection: SEQUENTIAL; DIRECT; or KEYED.

If the following three conditions are satisfied, *acc* is assigned the value SEQUENTIAL:

1. You are using either INQUIRE by unit or INQUIRE by unnamed file (where the unnamed file has a ddname of the form FTnnFmmm), and
2. The unit is preconnected, and
3. No OPEN, READ, or WRITE statement has been issued for the unit.

FORM=*frm*

frm is a character variable or character array element. It is assigned a value that corresponds to the type of file connection: FORMATTED or UNFORMATTED.

If the following three conditions are satisfied, *frm* is assigned the value FORMATTED:

1. You are using either INQUIRE by unit or INQUIRE by unnamed file (where the unnamed file has a ddname of the form FTnnFmmm), and
2. The unit is preconnected, and
3. No OPEN, READ, or WRITE statement has been issued for the unit.

RECL=*rcl*

rcl is a variable or integer array element of length 4 and is set to the record length of the file. The record length is measured in characters for formatted records, and in bytes for unformatted records.

NEXTREC=*nxr*

nxr is an integer variable or integer array element of length 4. *nxr* is set to the value of n + 1, where n is the record number of the last record read or written on the direct access file. If the file is connected, but no records have been read or written since the connection was established, *nxr* equals 1.

BLANK=*blk*

blk is a character variable or character array element. It is assigned the value NULL if blanks in arithmetic input fields are treated as blanks; ZERO if they are treated as zeros.

If you are using INQUIRE by unit or INQUIRE by unnamed file, and the unit is preconnected but no I/O statements other than INQUIRE, BACKSPACE, or REWIND have been issued for that unit, *blk* is assigned the value ZERO.

CHAR=*chr*

chr is a character variable or array element. If the file is connected, *chr* is assigned the value DBCS, if CHAR='DBCS' was specified on the OPEN statement; otherwise *chr* is assigned the value NODBCS. If the file is not connected, *chr* will become undefined.

ACTION=*act*

act is a character variable or character array element that is assigned one of the following values:

WRITE If the file is connected for writing records only

Note: For files connected for keyed access, *act* is assigned the value WRITE if the file is connected to load records into an empty file.

READ If the file is connected for reading records only

READWRITE
If the file is connected for reading and writing records

Note: For files connected for keyed access, *act* is assigned the value READWRITE if the file is connected to allow retrieval and update operations.

For all forms of the INQUIRE statement, a file must be connected for *act* to become defined.

WRITE=*wri*

wri is a character variable or character array element that is assigned the value YES if the file is connected only to have records written to it; otherwise, *wri* is assigned the value NO.

For files connected for keyed access, *wri* is assigned the value YES if the file is connected only to load records into the file; otherwise, *wri* is assigned the value NO.

READ=*ron*

ron is a character variable or character array element whose value is YES if the file is connected only for reading records. If the file is not connected only for reading, *ron* is assigned the value NO.

READWRITE=*rwr*

rwr is a character variable or character array element whose value is YES if the file is connected for reading and writing records. If the file is not connected for both reading and writing records, *rwr* is assigned the value NO.

For files connected for keyed access, *rwr* is assigned the value YES if the file is connected for both retrieval and update operations. If the file is not connected for both retrieval and update operations, *rwr* is assigned the value NO.

PASSWORD=*pwd*

pwd is a character expression of up to eight characters in length. You must specify the file's read password for *pwd*.

The PASSWORD= parameter is only necessary if the file is a VSAM file that was password-protected when it was defined with the Access Method Services program.

KEYID=*kid*

kid is an integer variable or integer array element of length 4. The value of *kid* is the relative position of the key of reference: that is, the key currently in use. (If the file is connected for keyed access, and if the OPEN statement for that file did not include a KEYS specifier, *kid* is assigned a value of 1.)

KEYLENGTH=*kle*

kle is an integer variable or integer array element of length 4.

If the file is connected for keyed access, the value of *kle* is the length of the leftmost character in the record of the key currently in use.

If the file is not connected for keyed access, but could be connected for keyed access, the value of *kle* is the position of the leftmost character in the record of the key of the file designated on the INQUIRE statement.

Otherwise, *kle* becomes undefined.

KEYSTART=*kst*

kst is an integer variable or integer array element of length 4.

If the file is connected for keyed access, the value of *kst* is the position of the leftmost character in the record of the key currently in use.

If the file is not connected for keyed access, but could be connected for keyed access, the value of *kst* is the position of the leftmost character in the record of the key of the file designated on the INQUIRE statement.

Otherwise, *kst* becomes undefined.

KEYEND=*ken*

ken is an integer variable or integer array element of length 4.

If the file is connected for keyed access, the value of *ken* is the position of the rightmost character in the record of the key currently in use.

If the file is not connected for keyed access, but could be connected for keyed access, the value of *ken* is the position of the leftmost character in the record of the key of the file designated on the INQUIRE statement.

Otherwise, *ken* becomes undefined.

LASTKEY=*lky*

lky is a variable or array element of any data type. If the file is connected for keyed access, *lky* is assigned the value of the key of the last keyed file record that was affected by a BACKSPACE, DELETE, READ, REWRITE, or WRITE statement. The length of *lky* should be at least as long as the key. If it is shorter than the key, the value of the key is truncated on the right; if it is longer, the value of the key is padded on the right with binary zeros.

LASTRECL=*lrl*

lrl is an integer variable or integer array element of length 4. If the file is connected for keyed access, *lrl* is assigned the length of the last keyed file record that was affected by a BACKSPACE, DELETE, READ, REWRITE, or WRITE statement.

If you are using this form of INQUIRE	And if no error occurs and these conditions are true	Then the variable or array elements of these specifiers become defined.
INQUIRE by Unit	Unit exists Unit is connected	NUMBER NAMED NAME[1] ACCESS SEQUENTIAL DIRECT KEYED FORM FORMATTED UNFORMATTED CHAR ACTION WRITE READ READWRITE
	Unit exists File connected for direct access	RECL NEXTREC
	Unit exists File connected for formatted I/O	BLANK
	Unit exists File connected for keyed access	KEYID KEYLENGTH KEYSTART KEYEND LASTKEY LASTRECL
INQUIRE by File	File is connected	NUMBER ACCESS FORM CHAR ACTION
	File exists	NAMED NAME SEQUENTIAL DIRECT KEYED FORMATTED UNFORMATTED WRITE READ READWRITE
	File exists File could be connected for direct access	RECL
	File exists File connected for direct access	NEXTREC
	File exists File connected for formatted I/O	BLANK
	File connected for keyed access	KEYID LASTKEY LASTRECL
	File exists File could be connected for keyed access	KEYLENGTH KEYSTART KEYEND

Figure 24 (Part 1 of 2). INQUIRE Optional Specifiers and Conditions under Which They Become Defined

If you are using this form of INQUIRE	And if no error occurs and these conditions are true	Then the variable or array elements of these specifiers become defined.
INQUIRE by Unnamed File	File is connected	NUMBER NAMED ACCESS SEQUENTIAL DIRECT KEYED FORM FORMATTED UNFORMATTED CHAR ACTION
	Unit and file exist	NAMED SEQUENTIAL DIRECT KEYED FORMATTED UNFORMATTED WRITE READ READWRITE
	Unit and file exist, OR File is connected File could be connected for direct access	RECL
	Unit and file exist File connected for direct access	NEXTREC
	File connected for formatted I/O	BLANK
	File is connected File is a keyed file	KEYID LASTKEY LASTRECL
	Unit and file exist File is a keyed file	KEYLENGTH KEYSTART KEYEND

Figure 24 (Part 2 of 2). INQUIRE Optional Specifiers and Conditions under Which They Become Defined

Note to Figure 24:

1 If there is a named file connected to the unit being inquired about, then the variable or array element given in the NAME specifier becomes defined.

Valid INQUIRE Statements:

```
INQUIRE (FILE=DDNAME, IOSTAT=IOS, EXIST=LEX, OPENED=LOD,
        NAMED=LNMD, NAME=FN, SEQUENTIAL=SEQ, DIRECT=DIR,
        FORMATTED=FMT, UNFORMATTED=UNF, ACCESS=ACC, FORM=FRM,
        NUMBER=INUM, RECL=IRCL, NEXTREC=INR, BLANK=BLNK)
```

```
INQUIRE (FILE='FT16K01',LASTRECL=RECL)
```

```
INQUIRE (0, IOSTAT=IACT(1), ERR=99999, EXIST=LACT(9),
        OPENED=LACT(8), NAMED=LACT(7), NAME=ACTUAL(1),
        SEQUENTIAL=ACTUAL(2), DIRECT=ACTUAL(3),
        FORMATTED=ACTUAL(4), UNFORMATTED=ACTUAL(5),
        ACCESS=ACTUAL(6), FORM=ACTUAL(7), NUMBER=IACT(2),
        RECL=IACT(3), NEXTREC=IACT(4), BLANK=ACTUAL(8))
```

```
INQUIRE (16,LASTKEY=LKEY,KEYSTART=START,KEYEND=END,
        KEYLENGTH=LENG)
```

```
INQUIRE (12,ACTION=ACT,KEYID=ID)

INQUIRE (FILE='/YOURFILE DATA *', NAME=NAM)

INQUIRE (FILE='/MYFILE OUTPUT A', EXIST=EX)

INQUIRE (FILE='/MYPROG.FORTDATA.PAYMENT', OPENED=OPN)

INQUIRE (FILE='/MYPDS.ACCOUNT(ABC)', NAMED=NMD)
```

INTEGER Type Statement

See "Explicit Type Statement" on page 91.

INTRINSIC Statement

The INTRINSIC statement identifies a name as representing a procedure supplied by VS FORTRAN Version 2 (a function or subprogram), and permits a specific intrinsic function name to be used as an actual argument.

Syntax

INTRINSIC *name1* **[**, *name2* ... **]**

name
is the generic or specific name of an intrinsic function.

The INTRINSIC statement is a specification statement and must precede statement function definitions and all executable statements.

Intrinsic functions are those functions known to the compiler. Intrinsic function names are either generic or specific. A generic name does not have a type, unless it is also a specific name.

Generic names simplify referring to intrinsic functions because the same function name may be used with more than one type of argument. Only a specific intrinsic function name may be used as an actual argument when the argument is an intrinsic function.

For the complete list of intrinsic function names and usage information for each function, Chapter 6, "Mathematical, Character, and Bit Routines" on page 263.

Appearance of a name in an INTRINSIC statement declares that name to be an intrinsic function name. If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit.

The following names of specific intrinsic functions must *not* be passed as actual arguments:

AMAX0
AMAX1
AMIN0
AMIN1
CHAR
DMAX1
DMIN1
FLOAT
ICHAR
IDINT
IFIX
INT
LGE
LGT
LLE
LLT
MAX0
MAX1
MIN0
MIN1
REAL
SNGL

CMPLX
DBLE
DBLEQ
DCMPLX
DFLOAT
DREAL
HFIX
IQINT
QCMPLX
QEXT
QEXTD
QFLOAT
QMAX1
QMIN1
QREAL
SNGLQ

The appearance of a generic function name in an INTRINSIC statement does not cause the name to lose its generic property. Only one appearance of a name in all the INTRINSIC statements of a program unit is permitted. The same name must not appear in both an EXTERNAL and an INTRINSIC statement in a program unit.

If the name of an intrinsic function appears in an explicit specification statement, the type must conform to its associated type.

If the name of an intrinsic function appears in the dummy argument list of a subprogram, that name is not considered as the name of an intrinsic function in that program unit.

Logical IF Statement

See "IF Statements" on page 127.

LOGICAL Type Statement

See "Explicit Type Statement" on page 91.

NAMELIST Statement

The NAMELIST statement specifies one or more lists of names for use in READ and WRITE statements.

Syntax

NAMELIST */name1/ list1* **[***/name2/ list2...* **]**

name

is a NAMELIST name. It is a name, enclosed in slashes, that must not be the same as a variable or array name.

list

is of the form *a1*, *a2*, ...

a

is a variable name or an array name.

The list of variables or array names belonging to a NAMELIST name ends with a new NAMELIST name enclosed in slashes or with the end of the NAMELIST statement. A variable name or an array name may belong to one or more NAMELIST lists.

Neither a dummy variable nor a dummy array name may appear in a NAMELIST list.

The NAMELIST statement must precede any statement function definitions and all executable statements. A NAMELIST name must be declared in a NAMELIST statement and may be declared only once. The name may appear only in input/output statements.

The NAMELIST statement declares a name *name* to refer to a particular list of variables or array names. Thereafter, the forms READ(*un*,*name*) and WRITE(*un*,*name*) are used to transmit data between the file associated with the unit *un* and the variables specified by the NAMELIST name *name*.

The rules for input/output conversion of NAMELIST data are the same as the rules for data conversion described in "General Rules for Data Conversion" on page 97. The NAMELIST data must be in a special form, described in "NAMELIST Input Data."

NAMELIST Input Data

To be read using a NAMELIST list, input data must be in a special form. The first character in each record to be read must be blank. The second character in the first record of a group of data records must be an ampersand (&) immediately followed by the NAMELIST name. The NAMELIST name must be followed by a blank and must not contain any embedded blanks. This name is followed by data items separated by commas. (A comma after the last item is optional.) The end of a data group is signaled by &END.

The form of the data items in an input record is:

- Name = Constant
  - The name may be an array element name or a variable name.
  - The constant may be integer, real, complex, logical, or character. (If the constants are logical, they may be in the form T or .TRUE. and F or .FALSE.; if the constants are characters, they must be included between apostrophes.)
  - Subscripts must be integer constants.
- Array Name = Set of Constants (separated by commas)
  - The *set of constants* consists of constants of the type integer, real, complex, logical, or character.
  - The number of constants must be less than or equal to the number of elements in the array.

- Successive occurrences of the same constant can be represented in the form *c***constant*, where *c* is a nonzero integer constant specifying the number of times the constant is to occur.

The variable names and array names specified in the input file must appear in the NAMELIST list, but the order is not significant. A name that has been made equivalent to a name in the input data cannot be substituted for that name in the NAMELIST list. The list can contain names of items in COMMON but must not contain dummy argument names.

Each data record must begin with a blank followed by a complete variable or array name or constant. Embedded blanks are not permitted in names or constants. Trailing blanks after integers and exponents are treated as zeros.

Examples:

All records have a blank in column 1, and begin in column 2.

```
&NAM1 I(2,3)=5,J=4,B=3.2
    .
    .
    .
A(3)=4.0,L=2,3,7*4,&END
```

where NAM1 is defined in a NAMELIST statement as:

```
NAMELIST /NAM1/A,B,I,J,L
```

and assuming that A is a 3-element array and I and L are 3X3 element arrays.

NAMELIST Output Data

When output data is written using a NAMELIST list, it is written in a form that can be read using a NAMELIST list.

- The data is preceded by &*name* and is followed by &END.
- All variable and array names specified in the NAMELIST list and their values are written out, each according to its type.
- Character data is included between apostrophes.
- The fields for the data are made large enough to contain all the significant digits.
- The values of a complete array are written out in columns.

OPEN Statement

An OPEN statement may be used to:

- Connect an existing file to a unit.
- Create a file that is preconnected.
- Create a file and connect it to a unit.
- Change certain specifiers of a connection between a file and a unit.

For more information on how to use the OPEN statement, see *VS FORTRAN Version 2 Programming Guide.*

Syntax

```
OPEN
     ( [UNIT=]un [, ERR=stl] [, STATUS=sta]
     [, FILE=fn] [, ACCESS=acc] [, BLANK=blk]
     [,CHAR=chr]
     [, FORM=frm] [, IOSTAT=ios]
     [, RECL=rcl][, ACTION=act]
     [, PASSWORD=pwd]
     [, KEYS=(start:end [, start:end] ... )] )
```

Each of the specifiers of the OPEN statement may appear only once. The unit identifier must be specified on the OPEN statement.

Before the OPEN statement is executed, the I/O unit specified by *un* may be either connected or not connected to an external file.

To connect a file for direct or keyed access or to connect any VSAM file, an OPEN statement is required.

The OPEN statement must not be used for internal files.

UNIT=*un*

specifies the external unit identifier. *un* is an integer expression of length 4 whose value must be either zero or positive.

un is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear immediately following the left parenthesis, and the other specifiers may appear in any order. If UNIT= is specified, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the OPEN statement. If an error is detected, control is transferred to *stl*. ERR=*stl* is optional.

STATUS=*sta*

sta is a character expression whose value (when any trailing blanks are removed) must be NEW, OLD, SCRATCH, or UNKNOWN. If STATUS is omitted, it is assumed to be UNKNOWN. STATUS=*sta* is optional.

The status of the external file can be specified as:

- NEW to create a file and connect it to a unit, or to create a preconnected file. (Successful execution of the OPEN statement changes the status to OLD.)

- OLD to connect an existing file to a unit.

- SCRATCH to connect an existing file, or to create and connect a new file, that will be deleted when it is disconnected. FILE=*fn* must not be specified; that is, SCRATCH must not be specified for a named file. If the file does not exist, STATUS='SCRATCH' opens the file as NEW. If the file does exist, STATUS='SCRATCH' opens the file as OLD.

- UNKNOWN to create and connect a new file, or to connect an existing file, of unknown status to a unit. If the file does not exist, STATUS='UNKNOWN' opens the file as NEW. If the file does exist, STATUS='UNKNOWN' opens the file as OLD.

Note:

The run-time options OCSTATUS and NOOCSTATUS affect the operation of an OPEN statement with either the STATUS='NEW' or STATUS='OLD'. For details on these options, see *VS FORTRAN Version 2 Programming Guide*.

FILE=*fn*

fn is a character expression used to identify a file. It must be preceded by FILE=. It may take one of the following formats:

- *ddname*

  a character expression whose value, when any trailing blanks are removed, is the name by which VS FORTRAN identifies the file definition. The expression must be 1 to 8 characters, the first one being a letter, #, or @, and the other seven being letters or digits, including # and @.

- **On VM:** */filename filetype* [*filemode*] [(*member-name*)]

  a character expression whose value, when any trailing blanks are removed, is the file name, file type, and file mode on VM.

  The slash (/) indicates that the file identifier is not a ddname; therefore, it must appear as the first character of the expression if this format is used.

  filename filetype
  each is a sequence of 1 to 8 characters chosen from letters, digits and special characters (hyphen (-), colon (:), underscore (_), #, @, and +).

  filemode
  from 1 to 2 characters whose first character ranges from A to Z and whose second character ranges from 0 to 6.

  - If the second character is not specified, 1 will be assumed.
  - If *filemode* is omitted, A will be used as the file mode.
  - If an asterisk (*) is specified for the file mode, the standard CMS search order will be used to find an existing file; for new files, the file will be created on the A disk.

member
: a member name, 1 to 8 characters long, and the parentheses are required to indicate that it is a member name. *member* may be used only when referring to a LOADLIB, MACLIB, or TXTLIB filetype. You may not use this form of the OPEN statement to update or create LOADLIB, MACLIB, or TXTLIB members.

Note: Lowercase letters are equivalent to uppercase letters. One or more blanks must separate each part of the CMS file identifier.

If this form is used, the FILEINF service routine may be used to set up the file characteristics prior to issuing the OPEN statement. If FILEINF is not used, the unit attribute table values will be used instead. For more information, see "FILEINF Subroutine" on page 283.

- **On MVS:** /*dsn* [(*member*)]

  a character expression whose value, when any trailing blanks are removed, is the data set name on MVS. It may be the data set name and member name for partitioned data sets.

  The slash (/) is used to indicate that the file identifier is not a ddname; therefore, it must appear as the first character of the expression if this form is used.

  dsn
  : a fully qualified MVS data set name. It consists of qualifiers, each of which are 1 through 8 characters long, separated by periods to a maximum length of 44 characters. Each qualifier may contain letters, numbers and special characters (#, and @). The first character must be a letter, # or @.

  member
  : a member within a partitioned data set. Partitioned data set names must be no longer than 54 characters (44 for the data set name and 8 for the member name plus the parentheses around the member name). Partitioned data sets are valid only for non-VSAM files.

  Note: Trailing blanks are ignored and lower case is equivalent to upper case.

  If this form is used, the FILEINF service routine may be used to set up the file characteristics prior to issuing the OPEN statement. If FILEINF is not used, the unit attribute table values will be used instead. For more information, see "FILEINF Subroutine" on page 283.

If the FILE specifier is omitted, a default ddname is assumed. Default ddnames must not be specified on the FILE specifier. Default ddnames take the following forms:

- FTnnFmmm, where nn is the two-digit unit identifier and mmm is the three-digit sequence number. Unnamed files connected for direct or sequential access are assigned default ddnames of this form.
- FTnnKkk, where nn is the two-digit unit identifier and kk is the two-digit sequence number. Keyed files are assigned default ddnames of this form.
- FTERRsss, where sss is the three-digit MTF subtask number. For the error message unit of an MTF subtask, the ddname defaults to this form.
- FTPRTsss, where sss is the three-digit MTF subtask number. For the PRINT/WRITE unit of an MTF subtask, the ddname defaults to this form.

(If the PRINT/WRITE unit and the error message unit are the same, the PRINT/WRITE unit's ddname is FTERRsss.)

In addition, the following default CMS file identifiers must not be specified on the FILE specifier:

- FILE FTnnFmmm, where nn is the two-digit unit identifier and mmm is the three-digit sequence number. Unnamed files connected by dynamic file allocation under CMS are assigned default CMS file identifiers of this form.
- FILE FTnnKkk, where nn is the two-digit unit identifier and kk is the two-digit sequence number. Unnamed files connected by dynamic file allocation for keyed access under CMS are assigned default CMS file identifiers of this form.

ACCESS=*acc*

acc is a character expression whose value (when any trailing blanks are removed) must be SEQUENTIAL, DIRECT, or KEYED. The values mean, respectively, that access to the file will be sequential, direct, or keyed. If ACCESS=*acc* is not specified, the file is connected for sequential access.

BLANK=*blk*

blk is a character expression whose value (when any trailing blanks are removed) must be either NULL or ZERO. This specifier affects the processing of the arithmetic fields accessed by READ statements with format specification or with list-directed only. It is ignored for nonarithmetic fields, for READ statements without format specification or with NAMELIST, and for all output statements. If NULL is specified, all blank characters in arithmetic formatted input fields on the specified unit are ignored, except that a field of all blanks has a value of zero. If ZERO is specified, all blanks, other than leading blanks, are treated as zeros. If the OPEN statement is specified, the default is NULL. If the OPEN statement is not specified, the default is ZERO. For information on how to control the treatment of blanks on a particular FORMAT statement, see the discussions of BN and BZ format codes under "BN Format Code" on page 114 and "BZ Format Code" on page 114, respectively. This specifier is only allowed for formatted I/O.

CHAR=*chr*

chr is a character expression whose value must be DBCS or NODBCS, when trailing blanks are removed. DBCS must be specified in order for bracketed double-byte characters to be interpreted correctly. Otherwise, these characters are seen as single-byte characters, with possible undesirable results. DBCS is required for the following:

- List-directed input that might have double-byte characters in character constants
- NAMELIST input that contains double-byte characters in character constants
- NAMELIST input with double-byte character names for variables or arrays
- Formatted I/O with run-time FORMAT statements that contain double-byte characters in character constants

The default is NODBCS.

FORM=*frm*

frm is a character expression whose value (when any trailing blanks are removed) must be FORMATTED or UNFORMATTED. This specifier indicates that the external file is being connected for formatted or unformatted input/output. If this specifier is omitted and ACCESS='SEQUENTIAL', a value of FORMATTED is assumed; otherwise, a value of UNFORMATTED is assumed.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. Its value is set to positive if an error is detected; it is set to zero if no error is detected. For VSAM files, return codes and reason codes are placed in *ios*.

ACTION=*act*

act indicates the kind of processing to be done to a file. It can be used with any files connected for sequential, direct, or keyed access. It is any character expression whose value is one of the following:

WRITE To indicate that a file is to have records written to it and that these records will not be read during the current connection. The file may or may not exist before the OPEN statement is issued. ACTION='WRITE' is not allowed under VM for library members.

For VSAM files, WRITE is used to open an empty file connected for keyed access for the loading of records. The records must be written in ascending key sequence.

READ To indicate that an existing file is to be read but not updated in any way. If both ACTION='READ' and STATUS='NEW' are specified, an error will be detected, independent of the OCSTATUS | NOOCSTATUS run-time options.

For files connected for keyed access, READ is used to open a non-empty file for retrieval. Update operations cannot be performed on the file.

READWRITE To indicate that a file may be both read from and written to during the current connection. The file may or may not exist before the OPEN statement is issued. ACTION='READWRITE' is not allowed under VM for library members.

For files connected for keyed access, READWRITE is used to open a file and make retrieval and update operations possible. For sequential and direct access files, you may execute WRITE statements in addition to READ statements. For keyed access files, you may execute REWRITE, DELETE, and WRITE statements in addition to READ statements. Using READWRITE, you can write to an empty keyed access file, and you need not write the records in ascending key sequence. READWRITE also enables you to open a keyed access file and then read from it to determine whether or not it contains any records.

If the ACTION specifier is omitted, the default for keyed access is READ. The default for sequential or direct access is READWRITE.

Specifier Used with ACCESS='DIRECT': The following specifier is used only if ACCESS='DIRECT' and **must** be specified for such access.

RECL=*rcl*

rcl is an integer expression of length 4. It specifies the record length of the file connected for direct access. The length is measured in characters for files consisting of formatted records and in bytes for files consisting of unformatted records.

Specifier Used for VSAM Files

PASSWORD=*pwd*

pwd specifies the password required to access a VSAM file, if the file was password-protected when it was defined with the Access Method Services program. It can be any character expression; however, if the character expression exceeds eight characters in length, only the first eight are used. If ACTION='READ', the file's read password is required; otherwise, its update password is required.

Specifier Used with ACCESS='KEYED'

KEYS=(*start***:***end* **[,** *start***:***end***] ...)**

gives the starting and ending positions, within keyed file records, of the primary and alternate-index keys to be used when accessing the keyed file.

start is an integer expression of length 4 whose value (which must be positive) represents the position in each record of a key's leftmost character.

end is an integer expression of length 4 whose value (which must be positive) represents the position in each record of the key's rightmost character. This value must not be less than the value of *start*.

The length of the key specified by a start-end pair is *end* - *start* + 1, and cannot exceed 255. Up to nine *start-end* pairs can be specified, each of which must have been defined with the Access Method Services program as the location of a key. If you have only one *start-end* pair to specify, you can omit the KEYS specifier; the missing information for the file is taken from the VSAM catalog. If you will be using multiple keys when accessing a keyed file, the KEYS specifier is necessary.

If the file is being loaded (ACTION='WRITE'), only the primary key can be specified.

Valid OPEN Statements:

```
OPEN (UNIT=2, IOSTAT=IOS, FILE='DDNAME', STATUS='NEW',
      ACCESS='SEQU'//'ENTIAL  ', FORM='FORMATTED',
      BLANK='ZERO')


OPEN (0, IOSTAT=IACT(1), FILE='DDNAME', STATUS='OLD',
      ACCESS='SEQUENTIAL', FORM='FORMATTED',
      BLANK='NULL')


OPEN (IOSTAT=IACT(1), STATUS='UNKNOWN', ACCESS='DIRECT',
      RECL=32, UNIT=IN+6)


OPEN (10,ACCESS='KEYED',ACTION='READWRITE')
OPEN (8,ACCESS='KEYED',KEYS=(2:7,15:22))

OPEN (1,FILE='/MYFILE OUTPUT')

OPEN (1,FILE='/YOURFILE DATA C4')

OPEN (1,FILE='/MYPROG.FORTDATA.PAYMENT')

OPEN (1,FILE='/MVSID.MYPDS.ACCOUNT(ABC)')
```

OPEN Statements for Non-Connected Units

Successful execution of the OPEN statement connects a unit to an external file with the specifiers designated (or assumed) in the OPEN statement. (For the specifiers allowed with the various definitions of data sets, see *VS FORTRAN Version 2 Programming Guide*.)

A unit may be connected in one program unit of an executable program. Once the unit is connected, it may be referenced from any other program unit in that executable program.

OPEN Statements for Connected Units

An OPEN statement for a unit that is already connected to an existing file allows you to change certain specifiers of a file and unit connection. The current file and unit connection (for the unit identified by the UNIT specifier) remains intact if there is no FILE specifier included on the OPEN statement.

If a unit is already connected to a file and you issue an OPEN statement for the same unit but a different file, the OPEN statement is executed as a CLOSE statement with no STATUS specifier followed by an OPEN.

In the instance of a preconnected unnamed file for which no file definition is given and no READ or WRITE statements have been issued, the properties specified by the OPEN statement become part of the connection.

For all other cases of connected units, only the BLANK specifier and the CHAR specifier may designate values different from the original connection.

Execution of the OPEN statement is affected by the run-time options, OCSTATUS and NOOCSTATUS. For more information on these options, and their effect on OPEN processing, see *VS FORTRAN Version 2 Programming Guide*.

Conditions that Prevent a File from being Connected:

- You specified an invalid unit identifier; that is a value that is outside the range of unit identifiers defined at your installation.
- You specified an invalid file name, a default ddname, or a default CMS file identifier on the FILE specifier.
- You specified invalid values; for example:
  - The value given in the STATUS specifier is inconsistent with the file existence property.
  - The value given in the RECL=*rcl* is not positive integer.
  - The OPEN statement specifies a different unit than the one to which the file is connected currently.

- – The KEYS specifier designates a start:end pair that does not represent a key available for use with the keyed file.

- ► OPEN processing encounters an error during file existence checking.

PARAMETER Statement

The parameter statement assigns a name to a constant.

Syntax

PARAMETER (*name1* = *c1* **[**, *name2* = *c2* ... **])**

name1 (name2...)
is a symbolic name. The name must be defined only once in a PARAMETER statement of a program unit.

c1 (c2...)
is a constant expression.

(Otherwise, the predefined conventions are used.) A PARAMETER statement may not precede an IMPLICIT NONE statement.

The type and length of a name of a constant must not conflict with subsequent specification statements, including IMPLICIT statements. The following is *invalid*:

```
PARAMETER   (INT=10)

IMPLICIT    CHARACTER*5(I)
```

If the length of a character constant represented by a name has been specified as an asterisk, the length is considered to be the length of the value of the character expression (*c1, c2*).

If the name (*name1, name2*) is of integer, real, double precision, or complex type, the corresponding expression (*c1, c2*) must be a constant arithmetic expression. The exponentiation operator is not permitted unless the exponent is of integer type.

If the name (*name1, name2*) is of character type, the corresponding expression (*c1, c2*) must be a character constant expression.

If the name (*name1, name2*) is of logical type, the corresponding expression (*c1, c2*) must be a logical expression containing only logical constants or names of logical constants.

Each (*name1, name2*) is the name of a constant that becomes defined with the value of the expression (*c1, c2*) that appears to the right of the equal sign. The value assigned is determined by the rules used for assignment statements (see Figure 21).

Any name of a constant that appears in an expression (*c1, c2*) must already be defined in this or a previous PARAMETER statement.

After it is defined, the name can be used in a subsequent expression or a DATA statement instead of the constant it represents. It must not, however, be part of a FORMAT statement or a format specification.

The name of a constant must not be used to form part of another constant; for example, any part of a complex constant.

Valid PARAMETER Statement:

```
CHARACTER*5 C1,C2
PARAMETER (C1='DATE ',C2='TIME ',RATE=2*1.414)
```

PAUSE Statement

The PAUSE statement temporarily halts the execution of the program and displays a message. The program waits until operator intervention causes it to resume execution. Program processing continues when the console operator presses the ENTER key.

Syntax

PAUSE [*n*]

n

a string of 1 through 5 decimal digits, or a character constant of up to 72 characters in length.

Valid PAUSE Statements:

```
PAUSE
PAUSE 20200
PAUSE 'MOUNT TEMPORARY TAPE.  TO RESUME, PRESS ENTER'
```

For the previous examples, the following messages would be displayed:

```
AFB001A PAUSE
AFB001A PAUSE 20200
AFB001A PAUSE MOUNT TEMPORARY TAPE.  TO RESUME, PRESS ENTER
```

PRINT Statements

The PRINT statement transfers data from internal storage to an external device.

Forms of the PRINT Statement:

1. "PRINT Statement—Formatted with Sequential Access"
2. "PRINT Statement—List-Directed I/O to External Devices" on page 161
3. "PRINT Statement—NAMELIST with External Devices" on page 162

Generally, each form of the PRINT statement has the same effect as that form of the WRITE statement. In comparison to the WRITE statement, the PRINT statement syntax is simpler, but its function is limited.

PRINT Statement—Formatted with Sequential Access

This statement transfers data from internal storage to an external device. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The unit used for the data transfer is installation dependent.

Syntax

PRINT *fmt* **[**,*list***]**

fmt

is a required format identifier. It can be one of the following:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression
- An array name

For explanations of these format identifiers, see "READ Statement—Formatted with Direct Access" on page 166.

list

is an I/O list. It can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. If the list is not present, the comma must be omitted. See "Implied DO in an Input/Output Statement" on page 81.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Data Transmission: A PRINT statement with FORMAT starts data transmission at the beginning of a record. The format codes in the format specification *fmt* are taken one by one and associated with every item of the list in the order they are specified. The data is taken from the item of the list, converted according to the corresponding format code, and the number of character data specified by the format code is transmitted onto the record of the external file. Data transmission stops when data has been taken from every item of the list.

If the list is not specified and the format specification starts with an I, E, F, D, G, L, Q, or Z, or is empty (that is, FORMAT( )), a blank record is written out.

The PRINT statement can be used to write over an end of file and extend the external file. An ENDFILE, BACKSPACE, CLOSE, or REWIND statement will then reinstate the end of file.

After execution of a sequential PRINT, no record exists in the file following the last record transferred by that statement.

For more information, see "WRITE Statement—Formatted with Sequential Access" on page 223.

PRINT Statement—List-Directed I/O to External Devices

This statement transfers data from internal storage to an external device. The type of the items specified in the I/O list determines the conversions to be performed. The unit used for the data transfer is installation dependent.

Syntax

PRINT * [,*list*]

*

an asterisk (*) specifies that a list-directed PRINT has to be executed.

list

is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 81.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Data Transmission: A PRINT statement with list-directed I/O accessing an external file starts data transmission at the beginning of a record. The data is taken from each item in the list in the order they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item in the list.

After execution of a sequential PRINT statement, no record exists in the file following the last record transferred by that statement.

The PRINT statement can write over an end of file and extend the external file. An ENDFILE, CLOSE, or REWIND statement will reinstate the end of file.

An external file with sequential access written with list-directed I/O is suitable *only* for printing, because a blank character is always inserted at the beginning of each record as a carriage control character.

For more information, see "WRITE Statement—List-Directed I/O to External Devices" on page 228.

PRINT Statement—NAMELIST with External Devices

This statement transfers data from internal storage to an external device. The type of the items specified in the NAMELIST statement determines the conversions to be performed. The unit used for the data transfer is installation dependent.

```
Syntax
PRINT name
```

name
is a NAMELIST name. See "NAMELIST Statement" on page 148.

Data Transmission: A PRINT statement with NAMELIST starts data transmission from the beginning of a record. The data is taken from each item in the NAMELIST with *name* in the order in which they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item in the NAMELIST name.

After execution of a PRINT statement with NAMELIST, no record exists in the file following the end of the NAMELIST just transmitted.

For more information, see "WRITE Statement—NAMELIST with External Devices" on page 233.

PROGRAM Statement

The PROGRAM statement assigns a name to a main program. It must be the first statement in the main program.

```
Syntax
PROGRAM name
```

name
is the name of the main program in which this statement appears.

A main program cannot contain any BLOCK DATA, SUBROUTINE, FUNCTION, or ENTRY statements.

A RETURN statement may appear; it has the same effect as a STOP statement.

The PROGRAM statement can only be used in a main program but is not required. If it is used, it must be the first statement of the main program. If it is not used, the name of the main program is assumed by this compiler to be MAIN.

The name must not be the same as any other name in the main program or as the name of a subprogram or common block in the same executable program. The name of a program does not have any type and the other specification statements have no effect on this *name*.

Execution of a program begins with the execution of the first executable statement of the main program. A main program may not be referred to from a subprogram or from itself.

READ Statements

READ statements transfer data from an external device to storage or from an internal file to storage.

Forms of the READ Statement:

1. "READ Statement—Asynchronous"
2. "READ Statement—Formatted with Direct Access" on page 166
3. "READ Statement—Formatted with Keyed Access" on page 169
4. "READ Statement—Formatted with Sequential Access" on page 173
5. "READ Statement—Formatted with Sequential Access to Internal Files" on page 176
6. "READ Statement—List-Directed I/O from External Devices" on page 179
7. "READ Statement—List-Directed I/O with Internal Files" on page 181
8. "READ Statement—NAMELIST with External Devices" on page 183
9. "READ Statement—NAMELIST with Internal Files" on page 186
10. "READ Statement—Unformatted with Direct Access" on page 187
11. "READ Statement—Unformatted with Keyed Access" on page 189
12. "READ Statement—Unformatted with Sequential Access" on page 193

READ Statement—Asynchronous

The asynchronous READ statement transmits unformatted data from a direct-access or tape device using sequential access. The asynchronous READ statement provides high-speed input. The statements are asynchronous because other program statements may be executed while data transfer is taking place. An OPEN statement is not permitted for asynchronous I/O.

Syntax

```
READ
    ( [UNIT=]un,
    ID=id )
    [list]
```

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the READ statement, all the specifiers can appear in any order.

ID=*id*

id is an integer constant or integer expression of length 4. It is the identifier for the READ statement.

list
is an asynchronous I/O list and may have any of four forms:

e
e1...e2
e1...
...e2

where:

e
is the name of an array.

e1 and *e2*
are the names of elements in the same array. The ellipsis (...) is an integral part of the syntax of the list and must appear in the positions indicated.

The unit specified by *un* must be connected to a file that resides on a direct-access or tape device. The array (*e*) or array elements (*e1* through *e2*) constitute the receiving area for the data to be read.

The asynchronous READ statement initiates a transmission. The WAIT statement, which must be executed for each asynchronous READ, ensures the conclusion of the transmission cycle. When executed after an asynchronous READ, the WAIT statement enables the program to refer to the transmitted data. This process ensures that a program will not refer to a data field while transmission to it is still in progress.

The asynchronous READ statement differs from other READ statements in that it requires the ID specifier which establishes a unique identification for the READ statement.

Synchronous READ statements may be executed for the file only after all asynchronous READ and WRITE operations have been completed and a REWIND has been executed for the file. Conversely, asynchronous READ statements may be executed for a file previously read synchronously after a REWIND or CLOSE has been executed.

Execution of an asynchronous READ statement initiates reading of the next record on the specified file. The record may contain more or less data than there are bytes in the receiving area. If there is more data, the excess is not transmitted to the receiving area; if there is less, the values of the excess array elements remain unaltered. The extent of the receiving area is determined as follows:

- If *e* is specified, the entire array is the receiving area. In this case, *e* may not be the name of an assumed-size array.
- If *e1...e2* is specified, the receiving area begins at array element *e1* and includes every element up to and including *e2*. The subscript value of *e1* must not exceed that of *e2*.
- If *e1...* is specified, the receiving area begins at element *e1* and includes every element up to and including the last element of the array. In this case, *e* may not be the name of an assumed-size array.
- If *...e2* is specified, the receiving area begins at the first element of the array and includes every element up to and including *e2*.

If *list* is not specified, there is no receiving area, no data is transmitted, and a record is skipped.

Subscripts in the list of the asynchronous READ must not be defined as array elements in the receiving area. If a function reference is used in a subscript, the function reference may not perform I/O on any file.

Given an array with elements of length *len*, transmission begins with the first *len* bytes of the record being placed in the first specified (or implied) array element. Each successive *len* byte of the record is placed in the array element with the next highest subscript value. Transmission ceases after all elements of the receiving area have been filled, or the entire record has been transmitted—whichever occurs first. If the record length is less than the receiving area size, the last array element to receive data may receive fewer than *len* bytes. If the record length is greater than the receiving area size, an error is detected.

The specified array may be multidimensional. Array elements are filled sequentially. Thus, during transmission, the leftmost subscript quantity increases most rapidly, and the rightmost least rapidly.

Any number of program statements may be executed between an asynchronous READ and its corresponding WAIT, subject to the following rules:

- No array element in the receiving area may appear in any such statement. This and the following rules apply also to indirect references to such array elements; that is, reference to or redefinition of any variable or array element associated by COMMON or EQUIVALENCE statements, or argument association with an array element in the receiving area.
- No executable statement may appear that redefines or undefines a variable or array element appearing in the subscript of *e*1 or *e*2. See "Valid and Invalid Programs" on page 3.
- If a function reference appears in the subscript expression of *e*1 or *e*2, the function may not be referred to by any statements executed between the asynchronous READ and the corresponding WAIT. Also, no subroutines or functions may be referred to that directly or indirectly refer to the function in the subscript reference, or to which the subscript function directly or indirectly refers.
- No function or subroutine may be executed that performs input or output on the file being manipulated, or that contains object-time dimensions that are in the receiving area (whether they be dummy arguments or in a common block).

Valid READ Statement:

```
READ (ID=10, UNIT=3*IN-3) ACTUAL(3)...ACTUAL(7)
```

READ Statement—Formatted with Direct Access

This READ statement transfers data from an external direct-access device into internal storage. A FORMAT statement (or a reference to a FORMAT statement) specifies the conversions to be performed during the transfer. The record to be read is identified by its relative record number. The data must reside on an external file that has been connected for direct access (see "OPEN Statement" on page 151).

Syntax

READ
  ([UNIT=]*un***,**
  [FMT=]*fmt***,**
  REC=*rec*
  [, ERR=*stl***]**
  [, IOSTAT=*ios***])**
  [*list***]**

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. statement. The other specifiers may appear in any order. If UNIT= is included on the READ statement FMT= must be used, and all the specifiers can appear in any order.

FMT=*fmt*

fmt is a required format identifier and can, optionally, be preceded by FMT=.

If FMT= is not specified, the format identifier must appear second. If both UNIT= and FMT= are included on the READ statement all the specifiers, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression
- An array name

The *statement label* must be the label of a FORMAT statement in the same program unit as the READ statement.

The *integer variable* must have been initialized by an ASSIGN statement with the label of a FORMAT statement. The FORMAT statement must be in the same program unit as the READ statement.

The *character constant* must constitute a valid format. The constant must be delimited by apostrophes, must begin with a left parenthesis, and end with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. An apostrophe in a constant enclosed in apostrophes is represented by two consecutive apostrophes.

The *character variable* and *character array element* must contain character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. Blank characters may precede the left parenthesis and character data may follow the right parenthesis. The length of the format identifier must not exceed the length of the array element.

The *character array name* must contain character data whose leftmost characters constitute a valid format identifier. The length of the format identifier may exceed the length of the first element of the array; it is considered the concatenation of all the array elements of the array in the order given by array element ordering.

The *array name* may be of type integer, real, double precision, logical, or complex.

The data must be a valid format identifier as described under "character array name" above.

The *character expression* may contain concatenations of character constants, character array elements, and character array names. Its value must be a valid format identifier. The operands of the expression must have length specifications that contain only integer constants or names of integer constants. (See Chapter 3, "Expressions" on page 31.)

REC=*rec*
rec is a relative record number. It is an integer expression whose value must be greater than zero. It represents the relative position of a record within the external file associated with *un*. The relative record number of the first record is 1. This specifier is required.

ERR=*stl*
stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected and is set to zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*.

list
is an I/O list. It can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 81.

An item in the list, or an item associated with it through EQUIVALENCE, COMMON, or argument passing, must not contain any portion of the format identifier *fmt*.

If this READ statement is encountered, the unit specified must exist and the file must be connected for direct access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit connection is established to an unnamed file. If the file is not preconnected, an error is detected.

This statement permits you to read records randomly from any location within an external file. It contrasts with the sequential input statements that process records, one after the other, from the beginning of an external file to its end. With the direct access statements, you can go directly to any record in the external file, process a record and go directly to any other record without having to process all the records in between.

Each record in a direct access file has a unique number associated with it. This number is the same as specified when the record is written. You must specify in the READ statement not only the external unit identifier, but also the number of the record to be read. Specifying the record number permits operations to be performed on selected records of the file instead of on records in their sequential order.

The OPEN statement specifies the size and form of the records in the direct access file. All the records of a file connected for direct access have the same length.

Data Transmission: A READ statement with FORMAT starts data transmission at the beginning of the record specified by REC=*rec*. The format codes in the format identifier *fmt* are taken one by one and associated with every item of the list in the order they are specified. The number of character data specified by the format code is taken from the record, converted according to the format code, and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item of the list or when the end of the record specified by *rec* is reached.

If the list is not specified and the format identifier starts with an I, E, F, D, G, L, Q or Z format code, or is empty (that is, FORMAT()), the internal record number is increased by one but no data is transferred.

Data and I/O List: The length of every record is specified in the RECL of the OPEN statement. If the record *rec* contains **more** data than is necessary to satisfy all the items of the list and the associated format identifier, the remaining data is ignored. If the record *rec* contains **less** data than is necessary to satisfy all the items of the list and the associated format identifier, an error is detected. If the format identifier indicates (for example, slash format code) that data be taken from the next record, then the internal record number *rec* is increased by one and data transmission continues with the next record. The INQUIRE statement can be used to determine the record number.

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315)

READ Statement—Formatted with Keyed Access

This READ statement transfers data from an external direct access device into internal storage. You specify in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The data must reside on an external file that has been connected for keyed access. (See "OPEN Statement" on page 151.)

There are two forms of this READ statement: the *direct retrieval keyed request* and the *sequential retrieval keyed request*. In a direct retrieval keyed request, you specify a full or partial key to be used in searching for the record to be retrieved.

In a sequential retrieval keyed request, you do not specify a key; the key of the record previously read or updated is used as the starting point and the next record in increasing key sequence is obtained. The key of reference from the previous I/O statement remains the key of reference for a sequential retrieval. If the file was just connected, the key of reference is the first key listed in the KEYS specifier of the OPEN statement, and the file is positioned before the first record with the lowest value for this key. A sequential retrieval keyed request reads this record.

Syntax for a Direct Retrieval Keyed Request

READ
    ([UNIT=]*un***, [FMT=]***fmt* **[, ERR=***stl***]**
    [, IOSTAT=*ios***] {, KEY=***key***|, KEYGE=***kge***|, KEYGT=***kgt***}**
    [, KEYID=*kid***] [, NOTFOUND=***stl***]) [***list***]**

Syntax for a Sequential Retrieval Keyed Request

READ
    ([UNIT=]*un***, [FMT=]***fmt***, [, ERR=***stl***]**
    [, IOSTAT=*ios***] [, NOTFOUND=***stl* **| , END=***stl***])**
    [*list***]**

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. statement. The other specifiers may appear in any order. If UNIT= is included on the READ statement, FMT= must be used and all the specifiers can appear in any order.

FMT=*fmt*

fmt is a required format identifier and can, optionally, be preceded by FMT=.

If FMT= is omitted, the format identifier must appear second. If both UNIT= and FMT= are included on the READ statement, all the specifiers, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression
- An array name

For explanations of these format identifiers, see "READ Statement—Formatted with Direct Access" on page 166.

ERR=*stl*
stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; negative if an end of file is detected; and is set to zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*.

KEY=*key***|KEYGE=***kge***|KEYGT=***kgt*
These specifiers cause a record to be retrieved by its key, and the file to be positioned at the end of the record. They supply a full or partial key value, which is used as a search argument.

KEY=*key* Specifies that the record to be retrieved is the first record whose key value is identical to the search argument. If the search argument is a partial key (shorter than the keys in the file), the record retrieved is the first one with a key whose leading part is identical to the partial key.

KEYGE=*kge* Specifies the following search criterion for the record to be retrieved: If the file contains a record whose key value is identical to *kge*, the first such record is retrieved. If not, the first record with the next greater key value is retrieved. If *kge* is a partial key (shorter than the keys in the file), the record retrieved is the first one with a key value whose leading part is equal to or greater than the partial key.

KEYGT=*kgt* Specifies that the record to be retrieved is the first one with a key value greater than *kgt*. If *kgt* is a partial key (shorter than the keys in the file), the record retrieved is the first one with a key value whose leading part is greater than the partial key.

key, *kge*, and *kgt* can be a character expression or a data item (a constant, variable, array element, or character substring) of an integer or character type whose length does not exceed the length of the key that is the target of the search. A shorter or partial key is called a generic key.

KEYID=*kid*

kid is an integer expression of length 4. Its value is the relative position of a start-end pair in the list of such pairs in the KEYS specifier of the OPEN statement. For example, KEYID=3 would designate the third start-end pair, and hence the third key, in the KEYS specifier. In this way, *kid* indicates which of multiple keys will be used to retrieve a record. The selected key, known as the "key of reference," remains in effect for all subsequent keyed access I/O statements until a different one is designated in another READ statement with a KEYID specifier.

If the KEYID specifier is omitted on the first READ statement for a file opened for keyed access, the first start-end pair on the KEYS specifier is used. If no KEYS specifier was given on the OPEN statement, KEYID must have a value of 1 or be omitted.

The KEYID specifier can be used only if the KEY, KEYGE, or KEYGT specifier is also used.

NOTFOUND=*stl*

stl is the statement label of an executable statement that is given control when a record-not-found condition occurs. For an explanation of this condition, see "Record Not Found," below.

END=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. When the end of the external file is encountered, control is transferred to *stl*.

This specifier can be used only in a sequential retrieval keyed request.

list

is an I/O list. It can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 81.

An item in the list, or an item associated with it through EQUIVALENCE, COMMON, or argument passing, must not contain any portion of the format identifier *fmt*.

Valid READ Statements:

```
READ (10,22,KEY='AC',NOTFOUND=97) AA,BB,CC
READ (UNIT=10,FMT=29,KEY='A01',NOTFOUND=32) AA, BB, CC
READ (10,29,KEYGE=CVAR,ERR=00) AA, BB, CC
READ (10,FMT=29,END=37) AA, BB, CC
READ (10,29) AA, BB, CC
READ (10,29,END=37) AA, BB, CC
READ (UNIT=10,FMT=29,NOTFOUND=87) AA, BB, CC
```

If the formatted keyed READ statement is encountered, the unit specified must exist and the file must have been connected for keyed access by means of an OPEN statement. The ACTION specifier on that OPEN statement must not have specified the value 'WRITE'. If the file is not so connected, an error is detected.

Data Transmission: For a direct retrieval keyed request, data transmission begins at the beginning of the record that satisfies the search criterion defined by the KEY, KEYGE, or KEYGT specifier. For a sequential retrieval keyed request, data transmission begins at the beginning of the record at which the file is currently positioned. The format codes in the format identifier *fmt* are taken one by one and associated with every item in the list in the order they are specified. The number and character data specified by the format code is taken from the record, converted according to the format code, and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item in the list or when the end of the record has been reached.

Data and I/O List: If the record contains **more** data than is necessary to satisfy all the items of the list and the associated format specification, the extra data is skipped over. The next sequential retrieval READ statement will start with the next sequential record. (This is the record with the next higher key value if the key value is unique or the next record with the same key if the key value is not unique.) If the record contains **less** data than is necessary to satisfy all the items of the list and the associated format identifier, an error is detected.

If the list is not specified and the format identifier starts with an I, E, F, D, G, L, Q, or Z format code or is empty (that is, FORMAT()), a record is skipped over.

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315)

End of File: Control is transferred to the statement specified by END when the end of the file is encountered; that is, when the file is already positioned at the end of the last record with the highest key value in the file and a sequential retrieval keyed request is issued. If IOSTAT=*ios* was specified, a negative integer value is assigned to *ios* when an end of file is detected. If ERR was specified but END was not, control passes to the statement specified by ERR when an end of file is detected. If neither END nor ERR was given, an error is detected.

Record Not Found: Control is transferred to the statement specified by NOTFOUND under one of these conditions:

- You made a direct retrieval keyed request, and *no record* in the file satisfied the search criterion defined by KEY, KEYGE, or KEYGT.
- You made a sequential retrieval keyed request, and there are *no more records* in which the leading portion of the key value is identical to the leading portion of the key value in the record retrieved by the last direct retrieval operation. The length of what is called the "leading portion of the key value" is equal to the length of the search argument (KEY=key, KEYGE=kge, or KEYGT=kgt) on the direct retrieval statement. This length may represent a full or partial key value.

The NOTFOUND specifier on the sequential retrieval keyed request is treated as an END specifier under any of these conditions:

- No direct retrieval keyed request has been made since the file was opened.
- The previous direct retrieval keyed request was unsuccessful.
- An operation that followed the previous direct retrieval keyed request did not successfully retrieve a record.
- A REWIND was issued after the previous direct retrieval keyed request.
- After the last direct retrieval request, a WRITE statement added a record whose key value differed in its leading positions from the key value being used in the comparison.

A record-not-found condition is not detected for a sequential retrieval keyed request that lacks a NOTFOUND specifier. In the absence of the NOTFOUND specifier, successive sequential retrieval requests may read records until the end of the file is reached.

If IOSTAT=*ios* was specified, a positive integer value is assigned to *ios* when a record-not-found condition is detected. If ERR is specified but NOTFOUND is not, control passes to the statement specified by ERR when a record-not-found condition is detected. If neither NOTFOUND nor ERR was given, an error is detected.

READ Statement—Formatted with Sequential Access

This READ statement transfers data from an external I/O device to storage. A FORMAT statement (or a reference to a FORMAT statement) specifies the conversions to be performed during the transfer. The data must reside in an external file that is connected for sequential access to a unit. (See "Input/Output Semantics" on page 46.)

The sequential I/O statements with format identifiers process records one after the other from the beginning of an external file to its end.

Syntax

READ *fmt* **[,** *list***]**

READ
    ([UNIT=]*un***, [FMT=]***fmt* **[, ERR=***stl***]**
    [, END=*stl***] [, IOSTAT=***ios***]) [***list***]**

UNIT=*un*

un is the external unit identifier. *un* is one of the following:

- An integer expression of length 4 whose value must be zero or positive, or
- An asterisk (*) representing an installation-dependent unit.

It is required in the first form of the READ statement and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order.

If UNIT= is included on the READ statement FMT= must be used and all the specifiers can appear in any order.

In the form of the READ in which *un* is not specified, *un* is installation dependent.

FMT=*fmt*

fmt is a required format identifier. It can, optionally, be preceded by FMT=.

If FMT= is omitted, the format identifier must appear second. If both UNIT= and FMT= are included on the READ statement, all the specifiers, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression
- An array name

For explanations of these format identifiers, see "READ Statement—Formatted with Direct Access" on page 166.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*.

END=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. When the end of the external file is encountered, control is transferred to *stl*.

When an end-of-file is encountered on a named file, the END= branch is taken, and the IOSTAT specifier, if present, is set to indicate an end-of-file. Under these conditions, the only I/O statements allowed are CLOSE, REWIND and BACKSPACE. If another READ is executed, message AFB217I is given, the END= branch is not taken, and the IOSTAT specifier is not set. The same sequence of instructions on a multiple file will cause the file to be positioned to the next subfile in sequence.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is detected, and zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*.

list

is an I/O list. It can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. (See "Implied DO in an Input/Output Statement" on page 81.) In the form of the READ where *un* is not specified, if the list is not present the comma must be omitted. An item in the list, or an item associated with it through EQUIVALENCE, COMMON, or argument passing, must not contain any portion of the format identifier *fmt*.

Valid READ Statements:

```
READ (5,98) A,B,(C(I,K),I=1,10)

READ (UNIT=2*IN-10, FMT='(19)', END=3600)

READ (10,22) AA,BB,CC
```

Invalid READ Statements:

```
READ ('(19)',08)        un must appear before fmt

READ ('(19)',UNIT=08)  FMT= must be used because UNIT= is specified.
```

When the NOOCSTATUS run-time option is in effect, the unit does not have to be connected to an external file for sequential access. For more information on the NOOCSTATUS option, see *VS FORTRAN Version 2 Programming Guide.*

Data Transmission: A READ statement with FORMAT starts data transmission at the beginning of a record. The format codes in the format identifier *fmt* are taken one by one and associated with every item of the list in the order they are specified. The number of character data specified by the format code is taken from the record, converted according to the format code, and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item of the list, or when the end of file is reached.

Data and I/O List: If the record contains **more** data than is necessary to satisfy all the items of the list and the associated format specification, the extra data is skipped over. The next READ statement with FORMAT will start with the next record if no other I/O statement is executed on that file. If the record contains **less** data than is necessary to satisfy all the items of the list and the associated format identifier, an error message will be issued.

If the list is not specified and the format identifier starts with an I, E, F, D, G, L, Q, or Z format code or is empty (that is, FORMAT()), a record is skipped over.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor transfer to be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315)

End of File: Control is transferred to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read into before the end of the file was encountered. If IOSTAT=*ios* is specified, a negative integer value is assigned to *ios*. Execution continues with the statement specified with END, if present, or with the next statement if END is not specified. If END and IOSTAT are both omitted, program execution is terminated when the end of the file is encountered.

READ Statement—Formatted with Sequential Access to Internal Files

This READ statement transfers data from one area of internal storage into another area of internal storage. A FORMAT statement (or a reference to a FORMAT statement) specifies the conversions to be performed during the transfer. The area in internal storage that is read from is called an internal file.

An internal file is always

- Connected to a unit
- Positioned before data transmission at the beginning of the storage area represented by the unit identifier

Syntax

READ
([UNIT=]*un***, [FMT=]***fmt* **[, ERR=***stl***]**
[, END=*stl***] [, IOSTAT=***ios***]) [***list***]**

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It can be the name of:

- A character variable
- A character array
- A character array element
- A character substring

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. If UNIT= is included in the READ statement, FMT= must be used and all the specifiers can appear in any order.

FMT=*fmt*

fmt is a required format identifier. It can, optionally, be preceded by FMT=.

The format identifier can be:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array name
- A character array element
- A character expression
- An array name

For explanations of these format identifiers, see "READ Statement—Formatted with Direct Access" on page 166.

The format specification must **not** be:

- In the area *un*, or

- Associated with *un* through EQUIVALENCE, COMMON, or argument passing.

If FMT= is omitted, the format specification must appear second. If both UNIT= and FMT= are included on the READ statement, all the specifiers, except *list*, can appear in any order.

ERR=*stl*
stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, transfer is made to *stl*.

END=*stl*
stl is the statement label of an executable statement in the same program unit as the READ statement. When the end of the storage area (*un*) is encountered, control is transferred to *stl*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

list
is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 81.

An item in the list must **not** be:

- Contained in the area represented by *un*, or
- Associated with any part of *un* through EQUIVALENCE, COMMON, or argument passing.

Valid READ Statements:

```
READ (5, 100) ((A(I,J),J=1,20),B(I),I=1,10)

READ (5, FMT=100) ((A(I,J),J=1,20),B(I),I=1,10)

READ (UNIT=5, FMT=100) ((A(I,J),J=1,20),B(I),I=1,10)
```

Invalid READ Statements:

```
READ (FMT=100, 5) ((A(I,J),J=1,20),B(I),I=1,10)       un must appear first because
                                                      UNIT= is not specified.

READ (100, UNIT=5) ((A(I,J),J=1,20),B(I),I=1,10)      FMT= must be used because
                                                      UNIT= is specified.
```

Data Transmission: An internal READ statement starts data transmission at the beginning of the storage area specified by *un*. The format codes in the format specification *fmt* are taken one by one and associated with every item of the list in the order they are specified. The number of character data specified by a format code is taken from the storage area *un*, converted according to the format code, and moved into the storage associated with the corresponding item in the list. Data transmission stops when data has been moved to every item of the list or when the end of the storage area is reached.

If *un* is a character variable, a character array element name, or a character substring name, it is treated as one record only in relation to the format identifier.

If *un* is a character array name, each array element is treated as one record in relation to the format identifier.

Data and I/O List: The length of a record is the length of the character variable, character substring name, or character array element specified by *un* when the READ statement is executed.

If a record contains **more** data than is necessary to satisfy all the items in the list and the associated format identifier, the remaining data is ignored.

If a record contains **less** data than is necessary to satisfy all the items in the list and the associated format identifier, an error is detected.

If the format identifier (for example, slash format code) indicates that further data is needed beyond the data contained in the character variable, character substring, or the last array element of a character array, an end of file is detected. If it is not the last array element in the character array, data is taken from the next array element.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

End of File: Control is transferred to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read into before the end of the file was encountered. If IOSTAT=*ios* is specified, a negative integer value is assigned to *ios* when an end of file is detected. Execution continues with the statement specified with END if present or with the next statement if END is not specified. If END and IOSTAT are both omitted, program execution is terminated when the end of the file is encountered.

Example:

```
  1  CHARACTER* 120 CHARVR
  2  READ (UNIT=5, FMT=100) CHARVR
100  FORMAT (A120)
  3  ASSIGN 200 TO J
  4  IF (CHARVR (3:4).EQ. 'AB') ASSIGN 300 TO J
  5  READ(UNIT = CHARVR, FMT=J) A1, A2, A3
200  FORMAT(4X,F5.1, F10.3, 3X, F12.8)
300  FORMAT (4X, F3.1, F6.3, 20X, F8.4)
```

Statement 1 defines a character variable, CHARVR, of fixed-length 120. Statement 2 reads into CHARVR 120 characters of input. Statement 3 assigns the format number 200 to the integer variable J. Statement 4 tests the third and fourth characters of CHARVR to determine which type of input is to be processed. If these two characters are AB, the format labeled 300 replaces the format labeled 200 and is used for processing the data. This is done by assigning statement label 300 to the integer variable J. Statement 5 reads from the internal file, CHARVR, and performs the conversion, using the appropriate FORMAT statement and assigning values to A1, A2, and A3.

READ Statement—List-Directed I/O from External Devices

This statement transfers data from an external device into internal storage. The type of the items specified in this statement determines the conversion to be performed. The data resides on an external file that is connected for sequential access to a unit (For a general discussion of file and unit connection, see "Input/Output Semantics" on page 46.)

Syntax

READ * [, *list***]**

READ
([UNIT=]*un***, [FMT=]* [, ERR=***stl***]**
[, END=*stl***] [, IOSTAT=***ios***])** **[***list***]**

UNIT=*un*

un is the external unit identifier. *un* is one of the following:

- An integer expression of length 4 whose value must be zero or positive
- An asterisk (*) representing an installation-dependent unit

un is required in the first form of the READ statement and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the READ statement, all the specifiers can appear in any order.

In the form of the READ in which *un* is not specified, *un* is installation dependent.

FMT=*

specifies that a list-directed READ is to be executed. It can, optionally, be preceded by FMT=.

If FMT= is omitted, the format identifier must appear second. If both UNIT= and FMT= are included on the READ statement, all the specifiers, except *list*, can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*.

END=*stl*
stl is the statement label of an executable statement in the same program unit as the READ statement. When the end of the external file is encountered, control is transferred to *stl*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

list
is an I/O list and can contain variable names, array element names, character substring names, array names (except names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 81.

Valid READ Statements:

```
READ (10,*) A,B,(C(I),I=1,4),D(4)
READ (10,FMT=*) A,B,(C(I),I=1,4),D(4)
READ (FMT=*,UNIT=10) A,B,(C(I),I=1,4),D(4)
READ (*,*) A,B,(C(I),I=1,4),D(4)
READ * A,B,(C(I),I=1,4),D(4)
READ (IOSTAT=IACT(1), UNIT=3*IN-2, FMT=*) ACTUAL(1)
```

Invalid READ Statements:

```
READ (FMT=*,10) A,B,(C(I),I=1,4),D(4)          un must appear first because
                                                   UNIT= is not specified.

READ (*,UNIT=10) A,B,(C(I),I=1,4),D(4)         FMT= must be used because
                                                   UNIT= is specified.

READ FMT=* A,B,(C(I),I=1,4),D(4)               FMT= must not be specified in the
                                                   second form of syntax.
```

If this READ statement is encountered, the unit specified by *un* must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language, and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

Data Transmission: A READ statement with list-directed I/O accessing an external file starts data transmission at the beginning of a record. One value on the external file is transferred to each item of the list in the order they are specified. The conversion to be performed depends on the type and length of the name of the item in the list. Data transmission stops when data has been transmitted to every item in the list, when a slash separator is encountered in the file or when the end of file is reached.

Data and I/O List: If the record contains more data than is necessary to satisfy all the items of the list, the extra data is skipped over. The next READ statement with list-directed I/O will start with the next record if no other I/O statement is executed on that file. If the record contains less data than is necessary to satisfy the list, and the record does not have a slash after the last element, an error is detected. If the list has not been satisfied when a slash separator is found, the remaining items in the list remain unaltered and execution of the READ is terminated.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

End of File: Control is transferred to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read before the end of the file was encountered. If IOSTAT=*ios* is specified, a negative integer value is assigned to *ios* when an end of file is detected. Execution continues with the statement specified with END, if present, or with the next statement if END is not specified. If END and IOSTAT are both omitted, program execution is terminated when the end of the file is encountered.

READ Statement—List-Directed I/O with Internal Files

This statement transfers data from one area of internal storage to one or more other areas of internal storage. The area in internal storage that is read from is called an internal file. The type of the items specified in this statement determines the conversion to be performed.

Syntax

```
READ
     ( [UNIT=]un, [FMT=]* [, ERR=stl]
     [, END=stl] [, IOSTAT=ios] ) [list]
```

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It can be the name of:

- A character variable
- A character array
- A character array element
- A character substring

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. If UNIT= is included on the READ statement, FMT= must be used and all the specifiers can appear in any order.

FMT=*

* specifies that a list-directed READ is to be executed. It can, optionally, be preceded by FMT=.

If FMT= is omitted, * must appear second. If both UNIT= and FMT= are included on the READ statement, all the specifiers, except *list*, can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*.

END=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. When the end of the storage area (*un*) is encountered, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected.

list

is an I/O list and can contain variable names, array element names, character substring names, array names (except names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 81.

Valid READ Statements:

```
READ (14,*) ACTUAL(1)
READ (14,FMT=*) ACTUAL(1)
READ (FMT=*,UNIT=14) ACTUAL(1)
READ (IOSTAT=IACT(1), UNIT=CHARVR, FMT=*) ACTUAL(1)
```

Data Transmission: An internal, list-directed READ statement starts data transmission at the beginning of the storage area specified by *un*. One value in the internal file is transferred to each item of the list in the order they are specified. The conversion to be performed depends on the type and length of the name of the item in the list. Data transmission stops when data has been moved to every item of the list or when the end of the storage area is reached.

If *un* is a character variable, a character array element name, or a character substring name, it is treated as one record. If *un* is a character array name, each array element is treated as one record.

Data and I/O List: The length of a record is the length of the character variable, character substring name, or character array element specified by *un* when the READ statement is executed.

If a record contains **more** data than is necessary to satisfy all the items in the list and the associated format identifier, the remaining data is ignored. The next READ statement with list-directed I/O will start with the next record if no other I/O statement is executed on that file.

If a record contains **less** data than is necessary to satisfy the list and the record does not have a slash after the last element, an error is detected. If the list has not been satisfied when a slash separator is found, the remaining items in the list remain unaltered and execution of the READ is terminated.

If the list indicates that more data items are to be moved and none remain in the character variable, character substring, or last array element of a character array, an end of file is detected. If an array element is not last and the list requires more data items than that element contains, the items are taken from the next array element.

End of File: Control is transferred to the statement specified by END when the end of the file is encountered; that is, when there is insufficient data in the character variable or array to satisfy the requirements of the I/O list. No indication is given of the number of list items read into before the end of the file was encountered. If IOSTAT=*ios* is specified, a negative integer value is assigned to *ios* when an end of file is detected. Execution continues with the statement specified with END if present or with the next statement if END is not specified. If END and IOSTAT are both omitted, program execution is terminated when the end of the file is encountered.

Example:

```
  1 CHARACTER* 50 CHARVR
  2 READ (UNIT=5, FMT=100) CHARVR
100 FORMAT (A50)
  3 READ (UNIT=CHARVR, FMT=*) A1, A2, A3
```

Statement 1 defines a character variable, CHARVR, of fixed-length 50. Statement 2 reads into CHARVR 50 characters of input. Statement 3 reads from CHARVR, performs the conversion (depending on the type and length of the names of the items in the list), and assigns values to A1, A2, and A3.

READ Statement—NAMELIST with External Devices

This statement transfers data from an external I/O device into storage. The type of the items specified in the NAMELIST determines the conversions to be performed. The data resides on an external file that is connected for sequential access to a unit (see "OPEN Statement" on page 151).

Syntax

READ *name*

READ
([UNIT=]*un*,
[FMT=]*name*
[, ERR=*stl***]**
[, END=*stl***]**
[, IOSTAT=*ios***])**

UNIT=*un*

un is the external unit identifier. *un* is one of the following:

- An integer expression of length 4 whose value must be zero or positive
- An asterisk (*) representing an installation-dependent unit

un is required in the first form of the READ statement and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the READ statement, all the specifiers can appear in any order.

In the form of the READ in which *un* is not specified, *un* is installation dependent.

FMT=*name*

name is a NAMELIST name. See "NAMELIST Statement" on page 148.

If FMT= is omitted, the NAMELIST name must appear second. If both UNIT= and FMT= are included in the READ statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*.

END=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. When the end of the external file is encountered, control is transferred to *stl*.

When an end-of-file is encountered on a named file, the END= branch is taken, and the IOSTAT specifier, if present, is set to indicate an end-of-file. Under these conditions, the only I/O statements allowed is CLOSE. If another READ is executed, message AFB217I is given, the END= branch is not taken, and the IOSTAT specifier is not set. The same sequence of instructions on a multiple file will cause the file to be positioned to the next subfile in sequence.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

Valid READ Statements:

```
READ (5,NEW_NAME)

READ (IN+IN+3, NAMEIN, IOSTAT=IOS)
```

Invalid READ Statements:

```
READ (NAMEX,5)          un must appear before name.

READ (5,NAMEX) A,B,C    list must not be specified.
```

If this type of READ statement (NAMELIST with external devices) is encountered, the unit specified by *un* must exist and it must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

The NAMELIST I/O statements associate the name given to the data in the program with the data itself. There is no format identifier, but the data is converted according to the type of data in the program. The data on the external file must be in a specific format. See "NAMELIST Input Data" on page 149.

The READ statement specifies the list of data to be transferred by referring to a NAMELIST statement.

BACKSPACE and REWIND should not be used with NAMELIST I/O. If they are, the results are unpredictable (see "BACKSPACE Statement" on page 57 and "REWIND Statement" on page 198).

Data Transmission: A READ statement with NAMELIST starts data transmission from the beginning of the NAMELIST with name *name* on the external file. The names associated with the NAMELIST name in the NAMELIST statement are matched with the names of the NAMELIST name on the external file. When a match is found, the value associated with the name on the external file is converted to the type of the name and transferred into storage. If a match is not found, an error is detected.

Data and NAMELIST: The NAMELIST name must appear on the external file. The variable names or array names associated with the NAMELIST name *name* in the NAMELIST statement must appear on the external file. They are read in the order they are specified in the NAMELIST statement, but they can appear in any order on the external file. (For the format of the input data, see "NAMELIST Input Data" on page 149.)

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

End of File: Control is transferred to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read before the end of the file was encountered. If END is omitted, program execution is terminated when the end of the file is encountered.

READ Statement—NAMELIST with Internal Files

This statement transfers data from one area of internal storage to one or more other areas of internal storage. The area of internal storage that is read from is called an internal file. The type of the items specified in an associated NAMELIST list determines the conversions to be performed.

Syntax

READ
([UNIT=]*un***,**
[FMT=]*name*
[, ERR=*stl***]**
[, END=*stl***]**
[, IOSTAT=*ios***])**

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It can be the name of:

- A character variable
- A character array
- A character array element
- A character substring

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. If UNIT= is included on the READ statement, FMT= must be used and all the specifiers can appear in any order.

FMT=*name*

name is a NAMELIST name. See "NAMELIST Statement" on page 148.

If FMT= is omitted, the NAMELIST name must appear second. If both UNIT= and FMT= are included on the READ statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*.

END=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. When the end of the external file is encountered, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected.

Valid READ Statements:

```
READ (12,NAME1)

READ (CHARVR, NAMEIN, IOSTAT=IOS)
```

The NAMELIST I/O statements associate the name given to the data in the program with the data itself. There is no format identifier, but the data is converted according to the type of data in the program. The data in the internal file must be in a specific format. See "NAMELIST Input Data" on page 149.

The READ statement specifies the list of data to be transferred by referring to a NAMELIST statement. This form of data transmission is useful for debugging purposes.

Data Transmission: A READ statement with NAMELIST starts data transmission at the beginning of the internal file specified by *un*. The data items associated with the NAMELIST name in the NAMELIST statement are matched with the values associated with the NAMELIST name in the internal file. When a match is found, the values associated with the name in the internal file are converted to the types of the data items in the NAMELIST list and assigned to the data items. If no match is found, an error is detected.

Data and NAMELIST: The NAMELIST name must appear in the internal file. The data items associated with the NAMELIST name in the NAMELIST statement must appear in the internal file. They are read in the order they are specified in the NAMELIST statement, but they can appear in any order in the internal file. (For the format of the input data, see "NAMELIST Input Data" on page 149.)

End of File: Control is transferred to the statement specified by END if:

- The NAMELIST input data in the internal file does not have an &END delimiter.
- The specified NAMELIST name is not in the internal file.

No indication is given of the number of list items read before control is transferred. If END is omitted, object program execution is terminated when the end of the internal file is encountered.

```
CHARACTER*40 CHARVR

NAMELIST /NL1/A,B,C

READ (CHARVR,NL1)
```

Assume CHARVR contains:

```
Position 2
v
&NL1 A=5,C=10,B=6,&END
```

Then A is assigned the value 5, B the value 6, and C the value 10.

READ Statement—Unformatted with Direct Access

This READ statement transfers data without conversion from an external direct-access device into internal storage. The data must reside on an external file that has been connected for direct access. (See "OPEN Statement" on page 151.) The record to be read is identified by a relative record number.

```
Syntax

READ
     ( [UNIT=]un, REC=rec [, ERR=stl]
     [, IOSTAT=ios]
     [, NUM=n] )
     [list]
```

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the READ statement, all the specifiers can appear in any order.

REC=*rec*

rec is a relative record number. It is an integer expression whose value must be greater than zero. It represents the relative position of a record within the external file associated with *un*. The relative record number of the first record is 1. This specifier is required.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*. ERR=*err* is optional.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected and zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*. IOSTAT=*ios* is optional.

NUM=*n*

n is an integer variable or an integer array element of length 4.

If NUM=*n* is specified, the variable or array element *n* is assigned a value representing the number of bytes of data transmitted to the elements specified by the I/O list.

Coding the NUM specifier suppresses the indication of an error that would occur if the number of data bytes represented by the I/O list is greater than the number of bytes in the record. In this case, *n* is set to a value that is the number of bytes in the record. Data from subsequent records is not read into the remaining I/O list items.

list

is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 81.

Data Transmission: A READ statement without format starts data transmission at the beginning of the record specified by REC=*rec*. The number of character data specified by the type of each item in the list is taken from the record and transmitted into the storage associated with the corresponding item in the list.

Data transmission stops when data has been transmitted to every item of the list.

If the list is not specified, the internal record number is increased by one but no data is transferred. The INQUIRE statement can be used to determine the record number.

Data and I/O List: The length of the records in the file is specified by RECL in the OPEN statement. If the record *rec* contains **more** data than is necessary to satisfy all the items of the list, the extra data is ignored. If the length of the record *rec* is *smaller* than the total amount of data needed to satisfy the items in the list, as much data as can be read from the record is read, and an error is detected unless the NUM specifier is given.

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315)

READ Statement—Unformatted with Keyed Access

This READ statement transfers data without conversion from an external direct-access I/O device into internal storage. The data must reside on an external file that has been opened for keyed access. (See "OPEN Statement" on page 151.)

There are two forms of this READ statement: the *direct retrieval keyed request* and the *sequential retrieval keyed request*. In a direct retrieval keyed request, you specify a full or partial key to be used in searching for the record to be retrieved.

In a sequential retrieval keyed request, you do not specify a key; the key of the record previously read or updated is used as the starting point and the next record in increasing key sequence is obtained. The key of reference from the previous I/O statement remains the key of reference for a sequential retrieval. If the file was just opened, the key of reference is the first key listed in the KEYS specifier of the OPEN statement, and the file is positioned before the first record with the lowest value for this key. A sequential retrieval keyed request reads this record.

Syntax for a Direct Retrieval Keyed Request

```
READ
     ( [UNIT=]un, [, ERR=stl] [, IOSTAT=ios]
     {, KEY=key | , KEYGE=kge | , KEYGT=kgt }[, KEYID=kid]
     [, NOTFOUND=stl] [, NUM=n] ) [list]
```

Syntax for a Sequential Retrieval Keyed Request

READ
    ([UNIT=]*un*, [, ERR=*stl*] [, IOSTAT=*ios*]
    [, NOTFOUND=*stl* | , END=*stl*] [, NUM=*n*])
    [*list*]

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the READ statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*.

KEY=*key*|KEYGE=*kge*|KEYGT=*kgt*

These specifiers cause a record to be retrieved by its key, and the file to be positioned at the end of the record. They supply a full or partial key value which is used as a search argument.

KEY=*key* Specifies that the record to be retrieved is the first record whose key value is identical to the search argument. If the search argument is a partial key (shorter than the keys in the file), the record retrieved is the first one with a key whose leading part is identical to the partial key.

KEYGE=*kge* Specifies the following search criterion for the record to be retrieved: If the file contains a record whose key value is identical to *kge*, the first such record is retrieved. If not, the first record with the next greater key value is retrieved. If *kge* is a partial key (shorter than the keys in the file), the record retrieved is the first one with a key value whose leading part is equal to or greater than the partial key.

KEYGT=*kgt* Specifies that the record to be retrieved is the first one with a key value greater than *kgt*. If *kgt* is a partial key (shorter than the keys in the file), the record retrieved is the first one with a key value whose leading part is greater than the partial key.

key, *kge*, or *kgt* can be a character expression or a data item (a constant, variable, array element, or character substring) of integer or character type whose length does not exceed the length of the key that is the target of the search. A shorter or partial key is called a generic key.

KEYID=*kid*

kid is an integer expression of length 4. Its value is the relative position of a start-end pair in the list of such pairs in the KEYS specifier of the OPEN statement. For example, KEYID=3 would designate the third start-end pair, and hence the third key, in the KEYS specifier. In this way, *kid* indicates which of multiple keys will be used to retrieve a record. The selected key, known as the "key of reference," remains in effect for all subsequent keyed access I/O statements until a different one is specified in another READ statement with a KEYID specifier.

If the KEYID specifier is omitted on the first READ statement for a file opened for keyed access, the first start-end pair on the KEYS specifier is used. If no KEYS specifier was given on the OPEN statement, KEYID must have a value of 1 or be omitted.

The KEYID specifier can be used only if the KEY, KEYGE, or KEYGT specifier is also used.

NOTFOUND=*stl*

stl is the statement label of an executable statement that is given control when a record-not-found condition occurs. For an explanation of this condition, see "Record Not Found," below.

END=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. When the end of the external file is encountered, control is transferred to *stl*.

This specifier can be used only on a sequential retrieval keyed request.

NUM=*n*

n is an integer variable or an integer array element of length 4.

If NUM=*n* is specified, the variable or array element *n* is assigned a value representing the number of bytes of data transmitted to the elements specified by the I/O list.

Coding the NUM specifier suppresses the indication of an error that would occur if the number of data bytes represented by the I/O list is greater than the number of bytes in the record. In this case, *n* is set to a value that is the number of bytes in the record. Data from subsequent records is not read into the remaining I/O list items.

list

is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 81.

Valid READ Statements:

```
READ (IOSTAT=IACT(1),UNIT=3*IN-2) ACTUAL(1)
READ (12,KEYGE=DEPTNO,NOTFOUND=86) DD,EE,FF
READ (UNIT=10,KEY='A01',NOTFOUND=32) AA, BB, CC
READ (10,KEYGT=CVAR,NUM=LENG) AA, (B(I),I=1, 100)
READ (10,END=37) AA, BB, CC
READ (10,NUM=LENG,NOTFOUND=87) AA, (B(I), I=1, 100)
```

If an unformatted keyed READ statement is encountered, the unit specified must exist and the file must have been connected for keyed access by means of an OPEN statement. The ACTION specifier on that OPEN statement must not have specified the value 'WRITE'. If the file is not so connected, an error is detected.

Data Transmission: For a direct retrieval keyed request, data transmission begins at the beginning of the record that satisfies the search criterion defined by the KEY, KEYGE, or KEYGT specifier. For a sequential retrieval keyed request, data transmission begins at the beginning of the record at which the file is currently positioned. The data specified by the item in the list is taken from the record and transmitted into the corresponding item in the list. Data transmission stops when data has been transmitted to every item in the list or when the end of file is reached.

If the list is not specified, a record is passed over and no data is transmitted.

Data and I/O List: If the record contains **more** data than is necessary to satisfy all the items in the list, the extra data is skipped over. The next sequential retrieval keyed request will start with the next sequential record. (This is the record with the next higher key value if the key value is unique or the next record with the same key if the key value is not unique.) If the record contains **less** data than is necessary to satisfy the list, an error is detected unless the NUM specifier was given.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315)

End of File: Control is transferred to the statement specified by END when the end of the file is encountered; that is, when the file is already positioned at the end of the last record with the highest key value in the file and a sequential retrieval keyed request was issued. If IOSTAT=*ios* was specified, a negative integer value is assigned to *ios* when an end of file is detected. If ERR was specified but END was not, control passes to the statement specified by ERR when an end of file is detected. If neither END nor ERR was given, an error is detected.

Record Not Found: Control is transferred to the statement specified by NOTFOUND under one of these conditions:

- You made a direct retrieval keyed request, and *no record* in the file satisfied the search criterion defined by KEY, KEYGE, or KEYGT.
- You made a sequential retrieval keyed request, and there are *no more records* in which the leading portion of the key value is identical to the leading portion of the key value in the record retrieved by the last direct retrieval operation. The length of what is called the "leading portion of the key value" is equal to the length of the search argument (KEY=key,

KEYGE=kge, or KEYGT=kgt) on the direct retrieval statement. This length may represent a full or partial key value.

The NOTFOUND specifier on the sequential retrieval keyed request is treated as an END specifier under any of these conditions:

- No direct retrieval keyed request has been made since the file was opened.
- The previous direct retrieval keyed request was unsuccessful.
- An operation that followed the previous direct retrieval keyed request did not successfully retrieve a record.
- A REWIND was issued after the previous direct retrieval keyed request.
- After the last direct retrieval request, a WRITE statement added a record whose key value differed in its leading positions from the key value being used in the comparison.

A record-not-found condition is not detected for a sequential retrieval keyed request that lacks a NOTFOUND specifier. In the absence of the NOTFOUND specifier, successive sequential retrieval requests may read records until the end of the file is reached.

If IOSTAT=*ios* was specified, a positive integer value is assigned to *ios* when a record-not-found condition is detected. If ERR is specified but NOTFOUND is not, control passes to the statement specified by ERR when a record-not-found condition is detected. If neither NOTFOUND nor ERR was given, an error is detected.

READ Statement—Unformatted with Sequential Access

This READ statement transfers data without conversion from an external I/O device into internal storage. The data resides on an external file that is connected for sequential access to a unit. (For a general discussion of file and unit connection, see "Input/Output Semantics" on page 46.)

The sequential I/O statements without format control process records one after the other from the beginning of an external file to its end.

Syntax

```
READ
    ( [UNIT=]un [, ERR=stl] [, END=stl]
    [, NUM=n][, IOSTAT=ios] )
    [list]
```

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the READ statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stl*.

END=*stl*

is the statement label of an executable statement in the same program unit as the READ statement. When the end of the external file is encountered, control is transferred to *stl*.

When an end-of-file is encountered on a named file, the END= branch is taken, and the IOSTAT specifier, if present, is set to indicate an end of file. Under these conditions, the only I/O statements allowed are CLOSE, REWIND, and BACKSPACE. If another READ is executed, message AFB217I is given, the END= branch is not taken, and the IOSTAT specifier is not set. The same sequence of instructions on an unnamed file will cause the file to be positioned to the next subfile in sequence.

NUM=*n*

n is an integer variable or an integer array element of length 4.

If NUM=*n* is specified, the variable or array element *n* is assigned a value representing the number of bytes of data transmitted to the elements specified by the I/O list.

Coding the NUM specifier suppresses the indication of an error that would occur if the number of data bytes represented by the I/O list is greater than the number of bytes in the record. In this case, *n* is set to a value which is the number of bytes in the record. Data from subsequent records is not read into the remaining I/O list items.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

list

is an I/O list. It can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 81.

If this READ statement is encountered, the unit specified by *un* must be connected to a file for sequential access. If the unit is not preconnected, an error is detected.

When the NOOCSTATUS execution time option is in effect, the unit does not have to be connected. For more information on the NOOCSTATUS option, see *VS FORTRAN Version 2 Programming Guide*.

Data Transmission: A READ statement without conversion starts data transmission at the beginning of a record. The data specified by the item in the list is taken from the record and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item of the list or when the end of file is reached.

If the list is not specified, a record is passed over without transmitting any data.

Data and I/O List: If the record contains more data than is necessary to satisfy all the items of the list, the extra data is skipped over. The next READ statement without format will start with the next record if no other I/O statement is executed on that file. If the record contains less data than is necessary to satisfy the list, an error is detected.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor transfer to be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

End of File: Control is transferred to the statement specified by END when the end of the file is encountered; that is, when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read into before the end of the file was encountered. If IOSTAT=*ios* is specified, a negative integer value is assigned to *ios* when an end of file is detected. Execution continues with the statement specified with END, if present, or with the next statement, if END is not specified. If END and IOSTAT are both omitted, program execution is terminated when the end of the file is encountered.

REAL Type Statement

See "Explicit Type Statement" on page 91.

RETURN Statement

The RETURN statement returns control to a calling program.

The RETURN statement can be used in either a **function** or a **subroutine** subprogram. A RETURN statement cannot terminate the range of a DO-loop.

RETURN Statement in a Function Subprogram

Function subprograms may contain RETURN statements. The RETURN statement signifies a logical conclusion of the computation and returns the computed function value and control to the calling program. (See "FUNCTION Statement" on page 120.)

Syntax

RETURN

Execution of a RETURN statement terminates the association between the dummy arguments of the subprogram and the current actual arguments. All entities (that is, common blocks, variables, or arrays) within the subprogram become undefined except:

- Entities specified in SAVE statements (see "SAVE Statement" on page 203)
- Entities given an initial value in a DATA or explicit specification statement and whose initial values were not changed
- Entities in a blank common block
- Entities in a named common block that appear in the subprogram and appear in at least one other program unit that is referring either directly or indirectly to the subprogram

All variables that are defined with a statement label become undefined regardless of whether the variable is in a common block or specified in a SAVE statement.

A function subprogram must not be referred to twice during the execution of an executable program without the execution of a RETURN statement in that subprogram. (See "END Statement" on page 83.)

RETURN Statement in a Subroutine Subprogram

Subroutine subprograms may contain RETURN statements. The RETURN statement signifies a logical conclusion of the computation and returns control to the calling program. (See "SUBROUTINE Statement" on page 208.)

Syntax

RETURN [*m*]

m

is an integer expression. If *m* is not specified in a RETURN statement, or if the value of *m* is less than one or greater than the number of asterisks in the SUBROUTINE or subroutine ENTRY statement that specifies the currently referenced name, control returns to the next statement following the CALL statement that initiated the subprogram reference. This completes the execution of the CALL statement.

If 1 ≤ *m* ≤ *n*, where *n* is the number of asterisks in the SUBROUTINE or subroutine ENTRY statement that specifies the currently referenced name, the value of *m* identifies the *m*th asterisk in the dummy argument list. There should be a one-to-one correspondence between the number of alternate return specifiers specified in the CALL statement and the number of asterisks specified in the SUBROUTINE statement or ENTRY statement dummy argument list. However, the alternate return specifiers need not be unique. Control is returned to the statement identified by the alternate return specifier in the CALL statement that is associated with the *m*th asterisk in the dummy argument list of the currently referenced name. This completes the execution of the CALL statement.

Execution of a RETURN statement terminates the association between the dummy arguments of the subprogram and the current actual arguments. All entities within the subprogram become undefined except:

- Entities specified in SAVE statements (see "SAVE Statement" on page 203)
- Entities given an initial value in a DATA or explicit specification statement and where initial values were not changed
- Entities in a blank common block
- Entities in a named common block that appear in the subprogram and appear in at least one other program unit that is referring either directly or indirectly to the subprogram

All variables that are defined with a statement label become undefined regardless of whether the variable is in a common block or specified in a SAVE statement.

A subprogram must not be referred to twice during the execution of an executable program without the execution of a RETURN statement in that subprogram.

A CALL statement that is used with a RETURN *m* form may be best understood by comparing it to a CALL and computed GO TO statement in sequence. For example, the following CALL statement:

```
CALL SUB (P,*20,Q,*35,R,*22)
```

is equivalent to:

```
CALL SUB (P,Q,R,I)
GO TO (20,35,22),I
```

where the index I is assigned a value of 1, 2, or 3 in the called subprogram.

REWIND Statement

The REWIND statement repositions a sequentially accessed file at the beginning of the first record of the file. The external file must be connected when you execute the statement. (See "OPEN Statement" on page 151.)

For a keyed file, the file must have been previously connected using an OPEN statement that specified an ACTION value of READ or READWRITE.

The REWIND statement positions the file to the beginning of the first record with the lowest value of the key of reference.

Syntax

REWIND *un*

REWIND
    ([UNIT=]*un*
    [, ERR=*stl***]**
    [, IOSTAT=*ios***])**

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and, if the second form of the statement is used, can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the READ statement, all the specifiers can appear in any order.

ERR=*stl*

is optional. *stl* is a statement label. If an error occurs in the execution of the REWIND statement, control is transferred to the statement labeled *stl*. That statement must be executable and must be in the same program unit as the REWIND statement. If ERR=*stl* is omitted, execution halts when an error is detected.

IOSTAT=*ios*

is optional. *ios* is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*.

If the unit specified by *un* is connected, it must be connected for sequential or keyed access. If the unit is not preconnected, an error is detected.

When the NOOCSTATUS run-time option is in effect, the unit does not need to be connected to an external file for sequential access. For more information on the NOOCSTATUS option, see *VS FORTRAN Version 2 Programming Guide*.

An external sequential file connected to the unit specified by *un* may or may not exist when the statement is executed. If the external sequential file does not exist, the REWIND statement has no effect. If the external sequential file does exist, an end-of-file is created, if necessary, and the file is positioned at the beginning of the first record.

For a sequential file, the REWIND statement causes a subsequent READ or WRITE statement referring to *un* to read data from or write data into the first record of the external file associated with *un*.

For a keyed file, a subsequent sequential retrieval keyed request will read the first record with the lowest key. The key of reference remains the same as it was before the REWIND statement was issued.

The REWIND statement may be used with asynchronous READ and WRITE statements provided that any input/output operation on the file has been completed by the execution of a WAIT statement. A WAIT statement is not required to complete the REWIND operation.

Transfer is made to the statement specified by the ERR specifier if an error is detected. If the IOSTAT=*ios* is specified, a positive integer value is assigned to *ios* when an error is detected. Then execution continues with the statement specified with the ERR specifier, if present, or with the next statement if ERR is not specified. If the ERR specifier and the IOSTAT specifier are both omitted, program execution is terminated when an error is detected.

Valid REWIND Statements:

```
REWIND (5)

REWIND (3*IN-2,ERR=99999)

REWIND (UNIT=2*IN+2)

REWIND (IOSTAT=IOS,ERR=99999,UNIT=2*IN-10)
```

REWRITE Statement—Formatted with Keyed Access

The REWRITE statement replaces a record in a keyed file. The record must have been retrieved by an immediately preceding sequential, direct, or keyed READ operation. No other operation, such as BACKSPACE or WRITE, can be issued for the same file between the READ and REWRITE statements.

For a keyed file, the file must have been previously connected, using an OPEN statement that specified an ACTION value of READWRITE.

Except for the key, any data in the retrieved record can be changed. If the records in the file have multiple keys, neither the value of the key being used for retrieval nor the value of the primary key can be changed.

Syntax

REWRITE
 ([UNIT=]*un***,**
 [FMT=]*fmt*
 [, ERR=*stl***]**
 [, IOSTAT=*ios***]**
 [, DUPKEY=*stl***])**
 list

UNIT=*un*

un is the external unit identifier. *un* must be an integer expression of length 4 whose value must be zero or positive.

un is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers can appear in any order. If UNIT= is included on the REWRITE statement, all the specifiers can appear in any order.

FMT=*fmt*

fmt is a format identifier. It can, optionally, be preceded by FMT=. If FMT=*fmt* is not specified, data transmission is defined by the items of the list. See "Data Transmission" on the following page.

If FMT is omitted, the format identifier must appear second. If both UNIT= and FMT= are included on the REWRITE statement, all the specifiers, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array element
- An array name
- A character expression

For explanations of these format identifiers, see "WRITE Statement—Formatted with Direct Access" on page 218.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the REWRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element It is set to positive if an error is detected; it is set to zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

DUPKEY=*stl*

stl is the statement label of a statement to which control is passed if a keyed record is being written and there is already a record in the file with the same key. This "duplicate key" condition can occur only if you tried to write a record containing a duplicate primary key or an alternate-index key that is defined to be unique.

list

is an I/O list. It can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. The list must represent all the data that is to comprise the new record, not just the fields that have been changed. The new copy of the record does not have to be the same length as the original; however, it must be long enough to include all the file's keys. (See "Implied DO in an Input/Output Statement" on page 81.) A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid REWRITE Statement:

```
REWRITE (12,15) AA,BB,CC
```

Data Transmission: A formatted REWRITE statement starts data transmission at the beginning of a record. The format codes in the format specification *fmt* are taken one by one and associated with every item of the list in the order they are specified. The data is taken from the item of the list, converted according to the corresponding format code, and the number of character data specified by the format code is transmitted onto the record of the external file. Data transmission stops when data has been taken from every item of the list.

If a transmission error is detected, control is transferred to the statement specified by ERR. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines can be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

REWRITE Statement—Unformatted with Keyed Access

The REWRITE statement replaces a record in a keyed file. The record must have been retrieved by an immediately preceding sequential, direct, or keyed READ operation. No other operation, such as BACKSPACE or WRITE, can be issued for the same file between the READ and REWRITE statements.

For a keyed file, the file must have been previously connected, using an OPEN statement which specified an ACTION value of READWRITE.

Except for the key, any data in the retrieved record can be changed. If the records in the file have multiple keys, neither the value of the key being used for retrieval nor the value of the primary key can be changed.

Syntax

REWRITE
    ([UNIT=]*un*
    [, ERR=*stl***]**
    [, IOSTAT=*ios***]**
    [, DUPKEY=*stl***]**
    [,NUM=*n***])**
    list

UNIT=*un*

un is the external unit identifier. *un* must be an integer expression of length 4 whose value must be zero or positive.

un is required and can, optionally, be preceded by UNIT= If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers can appear in any order. If UNIT= is included on the REWRITE statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the REWRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. It is set to positive if an error is detected; it is set to zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

DUPKEY=*stl*

stl is the number of a statement to which control is passed if a keyed record is being written and there is already a record in the file with the same key. This "duplicate key" condition can occur only if you tried to write a record containing a duplicate primary key or an alternate-index key that is defined to be unique.

NUM=*n*

n is an integer variable or an integer array element of length 4.

If NUM=*n* is specified, the variable or array element *n* is assigned a value representing the number of bytes of data transmitted to the elements specified by the I/O list.

Coding the NUM specifier suppresses the indication of an error that would occur if the number of bytes represented by the I/O list is greater than the number of bytes that can be written into the record. In this case, *n* is set to a value that is the maximum length record that can be written. Data from remaining I/O list items is not written into subsequent records.

list

is an I/O list. It can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. The list must represent all the data that is to comprise the new record, not just the fields that have been changed. The new copy of the record does not have to be the same length as the original; however, it must be long enough to include all the file's keys. (See "Implied DO in an Input/Output Statement" on page 81.) A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid REWRITE Statement:

```
REWRITE (12) AA,BB,CC
```

If the unit specified by *un* is connected, it must be connected for sequential access. If it is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

Data Transmission: An unformatted REWRITE statement without conversion starts data transmission at the beginning of a record. The data is taken from the items of the list in the order in which they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item of the list.

If a transmission error is detected, control is transferred to the statement specified by ERR. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines can be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

SAVE Statement

The SAVE statement retains the definition status of the name of a named common block, variable, or array after the execution of a RETURN or END statement in a subprogram.

Because VS FORTRAN Version 2 saves these names without user action, the SAVE statement serves only as a documentation aid.

Syntax

SAVE [*name1* **[,** *name2* **...]]**

name
: is a named common block name preceded and followed by a slash, a variable name, or an array name. Redundant appearances of an item are not permitted.

Dummy argument names, procedure names, and names of entities in a common block must not appear in a SAVE statement.

A SAVE statement without a list is treated as though it contained the names of all common items in that program unit.

The appearance of a named common block in a SAVE statement has the effect of specifying all entities in that named common block.

The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

- Entities specified by SAVE statements.
- Entities in a blank common block.
- Initially defined entities that have neither been redefined nor become undefined.
- Entities in named common blocks that appear in the subprogram and appear in at least one other program unit that is referring, either directly or indirectly, to that subprogram. The entities in a named common block may become undefined by execution of a RETURN or END statement in another program unit.

Within a function or subroutine subprogram, an entity (that is, a common block, variable, or array) specified by a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram.

If a local entity that is specified by a SAVE statement and is not in a common block is in a defined state at the time a RETURN or END statement is executed in a subprogram, that entity is defined with the same value at the next reference of that subprogram. An entity in a common block never becomes undefined as a result of the execution of a RETURN or END statement in a program unit that does not reference that common block. The entities in a named common block may become undefined or redefined by some other program unit.

Statement Function Statement

A statement function definition specifies operations to be performed whenever that statement function name appears as a function reference in another statement in the same program unit.

Syntax

name ([*arg1* [, *arg2* ...]) = *m*

name
is the statement function name (see "Names" on page 7).

arg
is a statement function dummy argument. It must be a distinct variable, that is, it may appear only once within the list of arguments.

m
is any arithmetic, logical, or character expression. Any statement function reference appearing in this expression must have been defined previously. In a function or subroutine subprogram, this expression can contain dummy arguments that appear in the FUNCTION, SUBROUTINE, or ENTRY statements that are previously defined within the same program unit. (For evaluation and restrictions of this expression, see Chapter 3, "Expressions" on page 31.)

A statement function definition is a nonexecutable statement. All statement function definitions to be used in a program must follow the specification statements and precede the first executable statement of the program.

The length of a character statement function must be an expression containing only integer constants or names of integer constants.

The dummy arguments enclosed in parentheses following the function name are dummy variables for which the arguments given in the function reference are substituted when the function reference is encountered. The same dummy arguments may be used in more than one statement function definition, and may be used as variables of the same type outside the statement function definitions, including dummy arguments of subprograms. The length specification of a dummy argument of type character must be an arithmetic expression containing only integer constants or names of integer constants.

An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument. It cannot be a character expression involving concatenation of one or more operands whose length specification is an asterisk.

If an actual argument is of character type, the associated dummy argument must be of character type and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument is greater than the length of an associated dummy argument, the leftmost characters of the actual argument are associated with the dummy argument.

The actual argument of a statement function reference must not be changed by the evaluation of the expression of that statement function. That is, an argument of a statement function cannot be modified by appearing in an external function reference that modifies it arguments.

The expression to the right of the equal sign defines the operations to be performed when a reference to this function appears in a statement elsewhere in the program unit. The expression defining the function must not contain (directly or indirectly) a reference to the function it is defining or a reference to any of the entry point names (PROGRAM, FUNCTION, SUBROUTINE, ENTRY) of the program unit where it is defined.

If the expression is an arithmetic expression, its type may be different from the type of the name of the function. Conversions are made as described for the assignment statement.

The name of a statement function must not appear in an EXTERNAL statement and must not be used as an actual argument.

For example, the statement:

```
FUNC(A,B) = 3.*A+B**2.+X+Y+Z
```

defines the statement function FUNC, where FUNC is the function name and A and B are the dummy arguments. The expression to the right of the equal sign defines the operations to be performed when the function reference appears in an arithmetic statement.

The function reference might appear in a statement as follows:

```
C = FUNC(D,E)
```

This is equivalent to:

```
C = 3.*D+E**2.+X+Y+Z
```

Notice the correspondence between the dummy arguments A and B in the function definition and the actual arguments D and E in the function reference.

Valid Statement Function Definitions and References:

Definition	Reference
SUM(A,B,C,D) = A+B+C+D	NET = GROS-SUM(TAX,COVER,HOSP,STOC)
	BIGSUM = SUM(A,B,SUM(C,D,E,F),G(I))
FUNC(Z) = A+X*Y*Z	ANS = FUNC(RESULT)
VALID(A,B) = .NOT. A .OR. B	VAL = TEST .OR. VALID(D,E)
VOLUME(R) = 4.0*PI/3.0 * R**3	TOTVOL = VOLUME(R1) + VOLUME(R2)

Invalid Statement Function Definitions:

SUBPRG(3,J,K)=3*I+J**3	Arguments must be variables.
SOMEF(A(I),B)=A(I)/B+3.	Arguments must not be array elements.
3FUNC(D)=3.14*E	Function name must begin with an alphabetic character.
BAD(A,B)=A+B+BAD(C,D)	A recursive definition is not permitted.
NOGOOD(A,A)=A*A	Arguments are not distinct variable names.
IFLUNK(I)=IDBLI(I)	Function IDBLI changes the value of argument I.

Invalid Statement Function References:

(The functions are defined as above.)

WRONG = SUM(TAX,COVER)	Number of arguments does not agree with above definition.
MIX = FUNC(I,J)	Types of the arguments do not agree with above definition.
MYGRAD(I)=IFLUNK(I)+I	I is modified by function IFLUNK(I).

Statement Labels

Statement labels identify statements in your source programs. Any statement can have a label, and may be written in either fixed form or free form. See "Source Language Statements" on page 8.

Fixed-Form Statement Labels

Fixed-form statement labels have the following attributes:

- They contain 1 to 5 decimal digits (not zero) and are on a non-continued line.
- Blanks and leading zeros are ignored.
- They are in columns 1 through 5.

Free-Form Statement Labels

Free-form statement labels have the following attributes:

- They must be the first nonblank characters (digits) on an initial line.
- Blanks and leading zeros are ignored.
- No blanks are needed between the statement label and the first nonblank character following.

See "ASSIGN Statement" on page 51.

STOP Statement

The STOP statement ends the processing of the object program and may display a message.

Syntax

STOP [*n*]

STOP ['*message*']

n
is a string of 1 through 5 decimal digits.

'*message*'
is a character constant enclosed in apostrophes and containing alphameric and/or special characters. Within the literal, an apostrophe is indicated by two successive apostrophes.

If you are running under MVS and you use a decimal value for the STOP statement, the value is returned to the job as the condition code for the job step being processed. If you are running under CMS in an EXEC and you use a decimal value for the STOP statement, the value is returned to your EXEC as the contents of variable &RETCODE.

When the program processes the STOP statement, operator message AFB0002I is displayed at the console.

Valid STOP Statements:

```
STOP
STOP 21212
STOP 'PROGRAM BACGAM EXECUTION COMPLETED'
```

A STOP statement cannot terminate the range of a DO-loop.

SUBROUTINE Statement

The SUBROUTINE statement identifies a subroutine subprogram.

Syntax

SUBROUTINE *name* **[(** *arg1* **[,***arg2...* **])]**

name
is the subroutine name. (See "Names" on page 7.)

arg
is a distinct dummy argument (that is, it may appear only once within the statement). There need not be any arguments, in which case the parentheses may be omitted. Each argument used must be a variable or array name, the dummy name of another subroutine or function subprogram, or an asterisk, where the character * denotes a return point specified by a statement label in the calling program.

Because the subroutine is a separate program unit, there is no conflict if the variable names and statement labels within it are the same as those in other program units.

The SUBROUTINE statement must be the first statement in the subprogram. The subroutine subprogram may contain any FORTRAN statement except a FUNCTION statement, another SUBROUTINE statement, a BLOCK DATA statement, or a PROGRAM statement. If an IMPLICIT statement is used in a subroutine subprogram, it must follow the SUBROUTINE statement and may only be preceded by another IMPLICIT statement, or a PARAMETER, FORMAT, or ENTRY statement.

The subroutine name must not appear in any other statement in the subroutine subprogram. It must not be the same as any name in the program unit or as the PROGRAM name, a subroutine name, or a common block name in any other program unit of the executable program. The subroutine subprogram may use one or more of its arguments to return values to the calling program. An argument so used will appear on the left side of an arithmetic, logical, or character assignment statement, in the list of a READ statement within the subprogram, or as an argument in a CALL statement or function reference that is assigned a value by the subroutine or function referred to.

The dummy arguments (*arg1*, *arg2*, *arg3*, ...) may be considered dummy names that are replaced at the time of execution by the actual arguments supplied in the CALL statement.

If a subroutine dummy argument is used as an adjustable array name, the array name and all the variables in the array declarators (except those in common) must be in the dummy argument list. See "Size and Type Declaration of an Array" on page 26.

The subroutine subprogram can be a set of commonly used computations, but it need not return any results to the calling program. For information about using RETURN and END statements in a subroutine subprogram, see "END Statement" on page 83 and "RETURN Statement" on page 196.

Actual Arguments in a Subroutine Subprogram

The actual arguments in a subroutine reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. The use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.

If an actual argument is of character type, the associated dummy argument must be of character type and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument is greater than the length of an associated dummy argument, the leftmost characters of the actual argument are associated with the dummy argument.

An actual argument in a subroutine reference must be one of the following:

- An expression, except for a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses (unless the operand is the name of a constant)
- An array name
- An intrinsic function name
- An external procedure name
- A dummy procedure name
- An alternate return specifier (statement label preceded by an asterisk)

An actual argument in a subroutine reference may be a dummy argument name that appears in a dummy argument list within the subprogram containing the reference. An asterisk dummy argument cannot be used as an actual argument in a subprogram reference.

Dummy Arguments in a Subroutine Subprogram

The dummy arguments of a subprogram appear after the subroutine name and are enclosed in parentheses. They are replaced at the time of execution of the CALL statement by the actual arguments supplied in the CALL statement in the calling program.

Dummy arguments must follow certain rules:

- None of the dummy argument names may appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, SAVE, INTRINSIC, or NAMELIST statement except as common block names.
- A dummy argument name must not be the same as the entry point name appearing in a PROGRAM, FUNCTION, SUBROUTINE, ENTRY, or statement function definition in the same program unit.

- The dummy arguments must correspond in number, order, and type to the actual arguments.
- If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, an array element, a substring, or an array. A constant, name of constant, subprogram name, or expression should not be written as an actual argument unless the programmer is certain that the corresponding dummy argument is not assigned a value in the subprogram.
- A referenced subprogram cannot assign new values to dummy arguments that are associated with other dummy arguments within the subprogram or with variables in common.
- The subprogram reserves no storage for the dummy argument, using the corresponding actual argument in the calling program for its calculations. Thus the value of the actual argument changes as soon as the dummy argument changes.

Valid SUBROUTINE statements:

1. Definition of subroutines SUB1 and SUB2: The following illustrates the two ways to define a subroutine with no dummy arguments.

   ```
   SUBROUTINE SUB1
   ⋮
   END

   SUBROUTINE SUB2()
   ⋮
   END
   ```

   The following are valid invocations of SUB1 and SUB2.

   ```
   CALL SUB1
   CALL SUB1()
   CALL SUB2
   CALL SUB2()
   ```

2. Definition of subroutine SUB3: The following illustrates an adjustable array and an explicitly dimensioned array as dummy arguments.

   ```
   SUBROUTINE SUB3(A, B, C)
   REAL A
   REAL B(*)
   REAL C(2, 5)
   ⋮
   END
   ```

   The sample invocations of SUB3 reference the following data declarations.

```
DIMENSION W(10), X(10), Z(5)
REAL Y

CALL SUB3(Y, W, X)       Call SUB3 with a variable and
                         2 array names

CALL SUB3(Z(3), X, W)    Call SUB3 with an array element
                         and 2 array names

CALL SUB3(2.5, W, X)     Call SUB3 with a constant and
                         2 array names

CALL SUB3(5*Y, X, W)     Call SUB3 with an expression and
                         2 array names
```

3. Definition of subroutine SUB4: The following illustrates the use of a logical variable as a dummy argument.

   ```
   SUBROUTINE SUB4(LOGL)
   LOGICAL LOGL
   ⋮
   END
   ```

   The sample invocations of SUB4 reference the following data declaration.

   ```
   LOGICAL L

   CALL SUB4(L)             Call using a logical variable

   CALL SUB4(.FALSE.)       Call using a logical constant

   CALL SUB4(X(5) .EQ. Y)   Call using a logical expression
   ```

4. Definition of subroutine SUB5: The following illustrates the use of a character variable of inherited length as a dummy argument.

   ```
   SUBROUTINE SUB5(CHAR)
   CHARACTER CHAR*(*)
   ⋮
   END
   ```

   The sample invocations of SUB5 reference the following variable declaration.

   ```
   CHARACTER*5 C1, C2

   CALL SUB5(C1)            Call using a character variable

   CALL SUB5(C1 // C2)      Call using a character expression
   ```

5. Definition of subroutine SUB6: The following illustrates subroutine and function subprogram names as dummy arguments.

   ```
   SUBROUTINE SUB6(SUBX, X, Y, FUNCX)
   Z = FUNCX(X, Y)
   CALL SUB7(SUBX)
   ⋮
   END
   ```

   The following shows the invocation of SUB6. The CALL passes a subroutine name and a function name.

```
EXTERNAL SUBA, FUNCA
⋮
CALL SUB6(SUBA, 1.0, 2.0, FUNCA)
```

6. Definition of subroutine SUB8: The following illustrates the use of * as dummy arguments.

```
SUBROUTINE SUB8(A, B, *, *, *)
⋮
IF(A .LT. 0.0) RETURN 1
IF(A .EQ. 0.0) RETURN 2
RETURN 3
END
```

The following shows the invocation of subroutine SUB8. The CALL passes statement numbers. Execution will continue at statement number 100, 200, or 300 if the return code is 1, 2, or 3 respectively. Otherwise, execution will continue at the statement after the call.

```
CALL SUB8(X(3), LOG(Z(2)), *100, *200, *300)
```

7. Definition of subroutine CLEAR: The following illustrates the use of an adjustable multidimensioned array.

```
     SUBROUTINE CLEAR (ARRY, M, N)
     INTEGER M, N, ARRY(M, N)
     DO 10 I = 1, M
     DO 10 J = 1, N
 10  ARRY(I,J) = 0
     RETURN
     END
```

The following is the invocation of CLEAR.

```
     INTEGER ARRAY1(10,15)
     CALL CLEAR(ARRAY1, 10, 15)
```

TRACE OFF Statement

The TRACE OFF statement stops the display of program flow by statement label.

Syntax

TRACE OFF

TRACE OFF may appear anywhere within a debug packet. After a TRACE ON statement, tracing continues until a TRACE OFF statement is encountered.

TRACE ON Statement

The TRACE ON statement initiates the display of program flow by statement label.

Syntax

TRACE ON

TRACE ON is executed only when the TRACE option appears in a DEBUG packet. (See "DEBUG Statement" on page 71.) Tracing continues until a TRACE OFF statement is encountered. TRACE ON stays in effect through any level of subprogram CALL or RETURN statement. However, if a TRACE ON statement is in effect and control is given to a program in which the TRACE option is not specified, the statement labels in that program are not traced.

Each time a statement with an external statement label is executed, a record of the statement label is made on the debug output file.

For a given debug packet, the TRACE ON statement takes effect immediately after the execution of the statement specified in the AT statement.

Unconditional GO TO

See "GO TO Statements" on page 125.

WAIT Statement

The WAIT statement synchronizes the completion of the data transmission begun by the corresponding asynchronous READ or WRITE statement.

Syntax

```
WAIT
     ([UNIT=]un,
     ID=id
     [, COND=i1]
     [, NUM=i2])
     [list]
```

UNIT=*un*

un is the external unit identifier. *un* is an unsigned integer expression of length 4.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the WAIT statement, all the specifiers can appear in any order.

ID=*id*

is required. *id* is an integer constant or integer expression of length 4, and is the identifier of a pending asynchronous READ or WRITE statement.

If the WAIT is completing an asynchronous READ, the expression *id* is subject to the following rules:

- No array element in the receiving area of the read may appear in the expression. This also includes indirect references to such elements; that is, reference to or redefinition of any variable or array element associated by a COMMON or EQUIVALENCE statement, or argument association with an array element in the receiving area.
- If a function reference appears in the subscript expression of *e*1 or *e*2, the function may not be referred to in the expression *id*. Also, no functions or subroutines may be referred to by the expression that directly or indirectly refers to the subscript function, or to which the subscript function directly or indirectly refers.

COND=*i1*

i1 is an integer variable name of length 4.

If COND=*i1* is specified, the variable *i1* is assigned a value of 1 if the input or output operation was completed successfully; 2 if an error condition was encountered; and 3 if an end-of-file condition was encountered while reading. In case of an error or end-of-file condition, the data in the receiving area may be meaningless.

NUM=*i2*

i2 is an integer variable name of length 4.

If NUM=*i2* is specified, the variable *i2* is assigned a value representing the number of bytes of data transmitted to the elements specified by the list. If the list requires more data from the record than the record contains, this specifier must be included on the WAIT statement. If the WAIT is completing an asynchronous WRITE, *i2* remains unaltered.

list

an asynchronous I/O list as specified for the asynchronous READ and WRITE statements.

If a list is included, it must specify the same receiving or transmitting area as the corresponding asynchronous READ or WRITE statement. It must not be specified if the asynchronous READ did not specify a list.

WAIT redefines a receiving area and makes it available for reference, or makes a transmitting area available for redefinition. The corresponding asynchronous READ or WRITE, which need not appear in the same program unit as the WAIT, is the statement that:

- Was not completed by the execution of another WAIT
- Refers to the same file as the WAIT
- Contains the same value for *id* in the ID=*id* form as did the asynchronous READ or WRITE when it was executed

The correspondence between WAIT and an asynchronous READ or WRITE holds for a particular execution of the statements. Different executions may establish different correspondences.

When the WAIT is completing an asynchronous READ, the subscripts in the list may not refer to array elements in the receiving area. If a function reference is used in a subscript, the function reference may not perform I/O on any file.

Valid WAIT Statements:

```
WAIT (8,ID=1) ARRAY(101)...ARRAY(500)

WAIT (9,ID=1,COND=ITEST)

WAIT (8,ID=1,NUM=N)

WAIT (9,ID=2)
```

WRITE Statements

WRITE statements transfer data from storage to an external device or from one internal file to another internal file.

Forms of the WRITE Statement:

1. "WRITE Statement—Asynchronous"
2. "WRITE Statement—Formatted with Direct Access" on page 218
3. "WRITE Statement—Formatted with Keyed Access" on page 220
4. "WRITE Statement—Formatted with Sequential Access" on page 223
5. "WRITE Statement—Formatted with Sequential Access to Internal Files" on page 225
6. "WRITE Statement—List-Directed I/O to External Devices" on page 228
7. "WRITE Statement—List-Directed I/O with Internal Files" on page 231
8. "WRITE Statement—NAMELIST with External Devices" on page 233
9. "WRITE Statement—NAMELIST with Internal Files" on page 235
10. "WRITE Statement—Unformatted with Direct Access" on page 236
11. "WRITE Statement—Unformatted with Keyed Access" on page 238
12. "WRITE Statement—Unformatted with Sequential Access" on page 240

WRITE Statement—Asynchronous

The asynchronous WRITE statement transmits data from an array in main storage to an external file. An OPEN statement is not permitted for asynchronous I/O.

Syntax

WRITE ([UNIT=]*un***, ID=***id* **)** *list*

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. If UNIT= is included on the WRITE statement, all the specifiers can appear in any order.

ID=*id*

id is an integer constant or integer expression of length 4. It is the identifier for the WRITE statement, and is used to identify the corresponding WAIT statement.

list

is an asynchronous I/O list that may have any of four forms:

e
e1...e2
e1...
...e2

e
is the name of an array.

e1 and *e2*
are the names of elements in the same array. The ellipsis (...) is an integral part of the syntax of the list and must appear in the positions indicated.

The unit specified by *un* must be connected to a file that resides on a direct-access or tape device. The array or array elements specified by *e* (or *e1* and *e2*) constitute the transmitting area for the data to be written. The extent of the transmitting area is determined as follows:

- If *e* is specified, the entire array is the transmitting area. In this case, *e* may not be the name of an assumed-size array.
- If *e1...e2* is specified, the transmitting area begins at array element *e1* and includes every element up to and including *e2*. The subscript value of *e1* must not exceed that of *e2*.
- If *e1...* is specified, the transmitting area begins at element *e1* and includes every element up to and including the last element of the array. In this case, *e* may not be the name of an assumed-size array.
- If *...e2* is specified, the transmitting area begins at the first element of the array and includes every element up to and including *e2*.
- If a function reference is used in a subscript of the list, the function reference may not perform I/O on any file.

Execution of an asynchronous WRITE statement initiates writing of a record on the specified file. The size of the record is equal to the size of the transmitting area. All the data in the area is written.

Given an array with elements of *len* length, the number of bytes transmitted will be *len* times the number of elements in the array. Elements are transmitted sequentially from the smallest subscript element to the highest. If the array is multidimensional, the leftmost subscript quantity increases most rapidly, and the rightmost least rapidly.

Any number of program statements may be executed between an asynchronous WRITE and its corresponding WAIT, subject to the following rules:

- No such statement may in any way assign a new value to any array element in the transmitting field. This and the following rules apply also to indirect references to such array elements; that is, assigning a new value to a variable or array elements associated by COMMON or EQUIVALENCE statements, or argument association with an array element in the transmitting area.
- No executable statement may appear that redefines or undefines a variable or array element appearing in the subscript of *e*1 or *e*2.
- If a function reference appears in the subscript expression of *e*1 or *e*2, the function may not be referred to by any statements executed between the asynchronous WRITE and the corresponding WAIT. Also, no subroutines or function may be referred to that directly or indirectly refer to the subscript function, or to which the subscript function directly or indirectly refers.

- No function or subroutine may be executed that performs input or output on the file being manipulated.

Valid WRITE Statement:

```
WRITE (ID=10, UNIT=2*IN+2) ... EXPECT(9)
```

WRITE Statement—Formatted with Direct Access

This statement transfers data from internal storage onto an external device. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The data must be sent to an external file that has been connected for direct access. (For a general discussion of file and unit connection, see "Input/Output Semantics" on page 46.)

Syntax

WRITE
([UNIT=]*un*,
[FMT=]*fmt*,
REC=*rec*
[, ERR=*stl***]**
[, IOSTAT=*ios***])**
[*list***]**

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the WRITE statement, FMT= must be used and all the specifiers can appear in any order.

FMT=*fmt*

fmt is a required format identifier. It can, optionally, be preceded by FMT=.

If FMT= is omitted, the format identifier must appear second. If both UNIT= and FMT= are included on the WRITE statement, all specifiers, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression
- An array name

For explanations of these format identifiers, see "READ Statement—Formatted with Direct Access" on page 166.

REC=*rec*

rec is an integer expression. It represents the relative position of a record within the file associated with *un*. Its value after conversion to integer, if necessary, must be greater than zero. The internal record number of the first record is 1. The INQUIRE statement can be used to determine the record number.

If *list* is omitted, a blank record is transmitted to the output device unless the FORMAT statement referred to contains, as its first specification, a character constant or slashes. In this case, the record or records indicated by these specifications are transmitted to the output device.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. Its value is positive if an error is detected, zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*.

list

is an I/O list and can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 81.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (REC=1, UNIT=11, FMT='(I9)')
```

```
WRITE (0, 1030, REC=N) NAME, ADDR, PHON
```

If this WRITE statement is encountered, the unit specified must exist and the file must be connected for direct access. If the file is not preconnected, an error is detected.

Data Transmission: A WRITE statement with FORMAT starts data transmission at the beginning of a record specified by REC=*rec*. The format codes in the format specification *fmt* are taken one by one and associated with every item of the list in the order they are specified. The data is taken from the item of the list, converted according to the corresponding format code, and the number of character data specified by the format code is transmitted onto the record of the external file. Data transmission stops when data has been taken from every item of the list, or when the end of the record specified by *rec* is reached.

If the list is not specified and the format specification starts with an I, E, F, D, G, L, Q, or Z or is empty (that is, FORMAT( )), the record is filled with blank characters and the relative record number *rec* is increased by one.

Data and I/O List: The length of every FORTRAN record is specified in the RECL specifier of the OPEN statement. If the length of the record *rec* is *greater* than the total amount of data specified by the format codes used during transmission of data, an error is detected, but as much data as can fit into the record is transmitted. If the length of the record *rec* is *smaller* than the total amount of data specified by the format codes used during transmission of data, an error is detected, but as much data as will fit in the record is transmitted. If the format specification indicates (for example, slash format code) that data be transmitted to the next record, then the relative record number *rec* is increased by one and data transmission continues.

Executing the WRITE statement causes the value of the NEXTREC variable in a preceding INQUIRE statement to be set to the relative record number of the last record written, increased by one. If an error is detected, the NEXTREC variable will contain the relative record number of the record being written.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be written; only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor transfer to be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

WRITE Statement—Formatted with Keyed Access

This statement transfers data from internal storage onto an external device. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The data must be sent to an external file that has been connected for keyed access.

Syntax

```
WRITE
      ( [UNIT=]un,
      [FMT=]fmt
      [, ERR=stl]
      [, IOSTAT=ios]
      [, DUPKEY=stl])
      list
```

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the WRITE statement, FMT= must be used and all the specifiers can appear in any order.

FMT=*fmt*

fmt is a required format identifier. It can, optionally, be preceded by FMT=.

If FMT= is omitted, the format identifier must appear second. If both

UNIT= and FMT= are included on the WRITE statement, all specifiers, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression
- An array name

For explanations of these format identifiers, see "READ Statement—Formatted with Direct Access" on page 166.

ERR=*stl*
stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. Its value is positive if an error is detected, zero if no error is detected. For VSAM files, return and reason codes are placed in *ios*.

DUPKEY=*stl*
stl is the statement label of an executable statement to which control is passed when a duplicate-key condition occurs. For an explanation of this condition, see "Duplicate Key," below.

list
is an I/O list and can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 81.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (10,18) AA,BB,CC
```

If this WRITE statement is encountered, the unit specified must exist and the file must have been connected for keyed access by means of an OPEN statement. The ACTION specifier of that OPEN statement must have specified the value 'READWRITE' or 'WRITE'. If the file is not so connected, an error is detected.

Data Transmission: If the WRITE statement was issued for a file connected by an OPEN statement with an ACTION specifier of 'WRITE', data transmission begins at the beginning of a new record. The new record will follow, in order of key value, the last record written. If the file was connected by an OPEN statement with an ACTION specifier of 'READWRITE', data transmission also begins at the beginning of a new record. In this case, however, the new record will be inserted following the record with a lower key value and preceding the record with a higher key value. If the new record has a key that is the same as a key already in the file, the new record is added following the last record with the same key. The format codes in the format specification *fmt* are taken one by one and associated with every item of the list in the order they are specified. The data is taken from the item of the list, converted according to the corresponding format code and the number of character data specified by the format code is transmitted onto a single record of the external file. Data transmission stops when data has been taken from every item of the list.

Data and I/O List: The amount of character data defined by all the format codes used during the transmission of the data defines the length of the record. A single WRITE statement can create only one record. The record must be long enough to include all the keys that are defined for the file.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be written; only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

Duplicate Key: Control is transferred to the statement specified by DUPKEY when a duplicate-key condition occurs; namely:

- The file is connected by an OPEN statement with an ACTION specifier of 'READWRITE', or when ACTION='WRITE', and
- An attempt was made to write a record with a key whose values must be unique, and
- The key value would have duplicated one that already exists for the same key in another record.

If IOSTAT=*ios* is specified, a positive integer value is assigned to *ios* when the duplicate-key condition is detected. If ERR is specified but DUPKEY is not, control passes to the statement specified by ERR when the duplicate-key condition is detected. If neither DUPKEY nor ERR was given, an error is detected.

Examples:

```
WRITE (UNIT=10,FMT=37) AA, BB, CC
WRITE (10,37) AA, BB, CC
WRITE (10,FMT=37,DUPKEY=77) AA, BB, CC
```

WRITE Statement—Formatted with Sequential Access

This statement transfers data from internal storage to a file. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) the conversions to be performed during the transfer. The data must be sent to an external file that is connected with sequential access to a unit. (For a general discussion of file and unit connection, see "Input/Output Semantics" on page 46.)

Syntax

```
WRITE
      ( [UNIT=]un,
      [FMT=]fmt
      [, ERR=stl]
      [, IOSTAT=ios] )
      [list]
```

UNIT=*un*

un is the external unit identifier. *un* is either:

- An integer expression of length 4 whose value must be zero or positive, or
- An asterisk (*) representing an installation-dependent unit.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the WRITE statement, all the specifiers can appear in any order.

FMT=*fmt*

fmt is a required format identifier. It can, optionally, be preceded by FMT=.

If FMT is omitted, the format identifier must appear second. If both UNIT= and FMT= are included on the WRITE statement, all specifiers, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression
- An array name

For explanations of these format identifiers, see "READ Statement—Formatted with Direct Access" on page 166.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

list

is an I/O list. It can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 81.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE(IOSTAT=IOS,ERR=99999,FMT=*,UNIT=2*IN+3)

WRITE(IN+8,NAMEOT,IOSTAT=IACT(1),ERR=99999) XRAY, CRYST, N, DELTA
```

If the unit specified by *un* is connected, it must be connected for sequential access. If the unit is not preconnected, an error is detected.

When the NOOCSTATUS run-time option is in effect, the unit does not need to be connected to an external file for sequential access. For more information on the NOOCSTATUS option, see *VS FORTRAN Version 2 Programming Guide.*

Data Transmission: A WRITE statement with FORMAT starts data transmission at the beginning of a record. The format codes in the format specification *fmt* are taken one by one and associated with every item of the list in the order they are specified. The data is taken from the item of the list, converted according to the corresponding format code, and the number of character data specified by the format code is transmitted onto the record of the external file. Data transmission stops when data has been taken from every item of the list.

If the list is not specified and the format specification starts with an I, E, F, D, G, L, Q or Z or is empty (that is, FORMAT( )), a blank record is written out.

The WRITE statement can be used to write over an end of file and extend the external file. An ENDFILE, BACKSPACE, CLOSE, or REWIND statement will then reinstate the end of file.

After execution of a sequential WRITE or PRINT, no record exists in the file following the last record transferred by that statement.

Data and I/O List: The amount of character data specified by all the format codes used during the transmission of the data defines the length of the FORTRAN record (also called a logical record). A single WRITE statement may create several FORTRAN records. This occurs when a slash format code is encountered in the format specification, or when the I/O list exceeds the format specification which causes the FORMAT statement to be used in full or part again. (See "FORMAT Statement" on page 96.)

VS FORTRAN Version 2 Programming Guide describes how to associate FORTRAN records (that is, logical records) and records in an external file.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be written; only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor transfer to be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

WRITE Statement—Formatted with Sequential Access to Internal Files

This statement transfers data from one or more areas in internal storage to another area in internal storage. It can be used to convert numeric data to character data and vice versa. The user specifies, in a FORMAT statement (or in a reference to a FORMAT statement), the conversions to be performed during the transfer. The receiving area in internal storage is called an internal file.

Syntax

```
WRITE
      ( [UNIT=]un,
      [FMT=]fmt
      [, ERR=stl]
      [, IOSTAT=ios] )
      [list]
```

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It can be the name of a character variable, character array, character array element, or character substring.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the WRITE statement, FMT= must be used, and all the specifiers can appear in any order.

FMT=*fmt*

is the format specification. It may, optionally, be preceded by FMT=.

If FMT= is omitted, the format specification must appear second. If both UNIT= and FMT= are included on the WRITE statement, all specifiers, except *list*, may appear in any order.

The format specification can be:

- The statement label of a FORMAT statement
- An integer variable
- A character constant
- A character variable
- A character array name
- A character array element
- A character expression
- An array name

For explanations of these format specifications, see "WRITE Statement—Formatted with Direct Access" on page 218.

ERR=*stl*
stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

list
is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 81.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Neither the format specification (*fmt*) nor an item in the list (*list*) can be:

- Contained in the area represented by *un*, or
- Associated with any part of *un* through EQUIVALENCE, COMMON, or argument passing.

Valid WRITE Statements:

```
CHARACTER *5 CHAR
DIMENSION IACT (10)

WRITE (3,100) (A(I),I=1,5),B

WRITE (3,FMT=100) (A(I),I=1,5),B

WRITE (FMT=100,UNIT=3) (A(I),I=1,5),B

WRITE (IOSTAT=IOS, ERR=99999, FMT='(A5)', UNIT=CHAR(1:5)) '1 2 3'

WRITE (CHAR(1:5), '(A5)', IOSTAT=IACT(1)) '4 5 6'
```

Invalid WRITE Statements:

```
WRITE (FMT=100,3) (A(I),I=1,5),B          un must appear first because
                                          UNIT= is not specified.

WRITE (100,UNIT=3) (A(I),I=1,5),B         FMT= must be used because UNIT=
                                          is specified.
```

Data Transmission: A WRITE statement starts data transmission at the beginning of the area specified by *un*. The format codes in the format specification *fmt* are taken one by one and associated with every item of the list in the order they are specified. Data is taken from the item of the list, converted according to the format code, and the number of character data specified by the format code is moved into the storage area *un*. Data transmission stops when data has been moved from every item of the list.

If *un* is a character variable, a character array element, or a character substring name, it is treated as one record only in relation to the format specification.

If *un* is a character array name, each array element is treated as one record in relation to the format specification.

If the list is not specified and the format specification starts with an I, E, F, D, G, L, Q, or Z or is empty (that is, FORMAT( )), the record is filled with blank characters and the relative record number *rec* is increased by one.

Data and I/O List: The length of a record is the length of the character variable, character substring name, or character array element specified by *un* when the WRITE statement is executed.

If the length of the record is greater than the amount of data specified by the items of the list and the associated format specification, the remainder of the record is filled with blank characters.

If the length of the record is less than the amount of data specified by the items of the list and the associated format specification, as much data as can fit in the record is transmitted and an error is detected.

The format specification may indicate (for example, slash format code) that data be moved to the next record of storage area *un*. If *un* specifies a character variable, a character array element, or a character substring name, an error is detected. If *un* specifies a character array name, data is moved into the next array element unless the last array element has been reached. In this latter case, an error is detected.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be written; only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

Valid Internal File Examples:

The following example illustrates how to use an internal READ of a character variable to initialize an integer array.

```
    CHARACTER*24 CHAR
    INTEGER IARRY(3,4)
```

Initialize the character variable CHAR.

```
    READ(5,'(A24)') CHAR
```

Assume that the data read into CHAR is:

```
  010203040506070809101112
```

Now, the program will use CHAR as an internal file and read it to initialize IARRY.

```
    READ (CHAR,10) ((IARRY(I,J), I = 1,3),J=1,4)
 10 FORMAT(12I2)
    WRITE (*) IARRY
    STOP
    END
```

The following example illustrates how to convert an integer number to its character representation. This example also illustrates a technique for changing a FORMAT statement dynamically; that is, the example initializes the specification of the field width for the A edit descriptor.

```
     CHARACTER*8 FMT
     DATA FMT /'(1X,AYY)'/
     I = 4
     WRITE (FMT(6:7), 10) I
 10  FORMAT (I2)
     ...
     WRITE (FMT) 'ABCD'
```

where YY can be any alphameric character, because YY is replaced by the character representation of the integer number.

WRITE Statement—List-Directed I/O to External Devices

This statement transfers data from internal storage to a file. The data must be sent to an external file that is connected with sequential access to a unit. (See "OPEN Statement" on page 151.) The type of the items specified in the statement determines the conversion to be performed.

Syntax

WRITE
([UNIT=]*un***,**
[FMT=]*
[, ERR=*stl***]**
[, IOSTAT=*ios***])**
[*list***]**

UNIT=*un*

un is the external unit identifier. *un* is either:

- An integer expression of length 4 whose value must be zero or positive, or
- An asterisk (*) representing an installation-dependent unit.

It is required and can, optionally, be preceded by UNIT=.

If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the WRITE statement, FMT= must be used, and all the specifiers, except *list*, can appear in any order.

FMT=*

An asterisk (*) specifies that a list-directed WRITE has to be executed. It can, optionally, be preceded by FMT= if *un* is specified.

If FMT= is omitted, the format identifier must appear second. If both UNIT= and FMT= are included on the WRITE statement, all specifiers, except *list*, may appear in any order.

ERR=*stl*
stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

list
is an I/O list and can contain variable names, array elements, character substring names, array names (except names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 81.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (30,*) REDUCT, INDUCT

WRITE (30,FMT=*) DEDUCT, RAINDUCT

WRITE (FMT=*,UNIT=30) MYDUCK, YOURDUCK

WRITE (5,*)

WRITE (FMT=*,UNIT=*) DAFFY,DUCK

WRITE (IOSTAT=IOS, ERR=99999, FMT=*, UNIT=2*IN+3)
      ''''//EXPECT(1)//''''
```

Invalid WRITE Statements:

```
WRITE (*,23) DONALD,DUCK          un must appear first because
                                     UNIT= is not specified.

WRITE (FMT=*,23) FIFTY5,ISEG      un must appear first because
                                     UNIT= is not specified.

WRITE (*,UNIT=*) FIFTY5,ISEG    FMT= must be used because
                                     UNIT= is specified.
```

If the unit specified by *un* is encountered, it must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

Data Transmission: A WRITE or PRINT statement with list-directed I/O accessing an external file starts data transmission at the beginning of a record. The data is taken from each item in the list in the order they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item in the list.

After execution of a sequential WRITE or PRINT statement, no record exists in the file following the last record transferred by that statement.

The WRITE or PRINT statement can write over an end of file and extend the external file. An ENDFILE, CLOSE, or REWIND statement will reinstate the end of file.

An external file with sequential access written with list-directed I/O is suitable *only* for printing, because a blank character is always inserted at the beginning of each record as a carriage control character.

Data and I/O List: The amount of character data specified by the items in the list and the necessary data separators define the length of the FORTRAN record (also called a logical record). A single WRITE or PRINT statement creates only one FORTRAN record.

For information on how to calculate the size of a record needed to hold all the converted list items, see Figure 25. It shows the width of the written field for any item's data type and length. The size of the record will be the sum of the field widths plus a byte to separate each field.

Data Type	Length	Field Width
Real	16	42 bytes
Real	8	25 bytes
Real	4	16 bytes
Logical	1 or 4	1 byte
Integer	2	6 bytes
Integer	4	11 bytes
Complex	32	84 bytes
Complex	16	51 bytes
Complex	8	25 bytes
Character	*	132 bytes (See Note)

Figure 25. Field Widths Needed for Data Types of Various Lengths

Note to Figure 25:

The number of bytes printed out is determined by the size of the character type item. The number of characters per record is determined by the type of data set being written to. The number of bytes per record is determined by the logical record length. For output that is sent to a terminal, a carriage control character is deleted at the beginning of each record. This is also true for a file defined with a carriage control character. Character data can be split between records. Numeric data cannot be split between records.

VS FORTRAN Version 2 Programming Guide describes how to associate FORTRAN records (that is, logical records) and records in a file. In particular, a logical record may span many physical records. A character constant or a complex constant can be split over the next physical record if there is not enough space on the current physical record to contain it all.

Character constants produced:

- Are not delimited by apostrophes
- Are not preceded or followed by any separators (including blanks)
- Have each internal apostrophe represented externally by one apostrophe

- Have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be written; only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

WRITE Statement—List-Directed I/O with Internal Files

This statement transfers data from one or more areas of internal storage to another area of internal storage. The receiving area is called an internal file. This statement can be used to convert numeric data to character data. The type of the items specified in the statement determines the conversion to be performed.

Syntax

```
WRITE
      ( [UNIT=]un,
      [FMT=]*
      [, ERR=stl]
      [, IOSTAT=ios] )
      [list]
```

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It can be the name of:

- A character variable
- A character array
- A character array element
- A character substring

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. If UNIT= is included on the WRITE statement, FMT= must be used and all the specifiers can appear in any order.

FMT=*

* specifies that a list-directed WRITE is to be executed. It can, optionally, be preceded by FMT=.

If FMT= is omitted, * must appear second. If both UNIT= and FMT= are included on the WRITE statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected.

list

is an I/O list and can contain variable names, array element names, character substring names, array names (except names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 81.

Valid WRITE Statements:

```
WRITE (28, *) FIFTY5,ISEG
WRITE (28, FMT=*) FIFTY5,ISEG
WRITE (FMT=*,UNIT=28) FIFTY5,ISEG
WRITE (IOSTAT=IACT(1), UNIT=CHARVR, FMT=*) ACTUAL(1)
```

Data Transmission: An internal WRITE statement starts data transmission at the beginning of the storage area specified by *un*. Each item of the list is transferred to the internal file in the order it is specified. The conversion to be performed depends on the type and length of the name of the item in the list. Data transmission stops when every item of the list has been moved to the internal file or when the end of the internal file is reached.

Data and I/O List: If *un* is a character variable, a character array element name, or a character substring name, it is treated as one record. If *un* is a character array name, each array element is treated as one record. If a record is not large enough to hold all the converted items, a new record is started for any noncharacter item that will exceed the record length. For character items, as much as can be put in the record is written there, and the remainder is written at the beginning of the next record.

The length of a record is the length of the character variable, character substring name, or character array element specified by *un* when the WRITE statement is executed.

For information on how to calculate the size of a record needed to hold all the converted list items, see Figure 25 on page 230. It shows the width of the written field for any item's data type and length. The size of the record will be the sum of the field widths plus a byte to separate each field.

```
    CHARACTER* 120 CHARVR

    WRITE (UNIT=CHARVR, FMT=*) A1, A2, A3

100 FORMAT (A120)

    WRITE (UNIT=6, FMT=100) CHARVR
```

Statement 1 defines a character variable, CHARVR, of fixed-length 120. Statement 2 writes the internal file represented by CHARVR by converting the values in A1, A2, and A3. Statement 3 writes the 120 characters of output onto an external file.

WRITE Statement—NAMELIST with External Devices

This statement transfers data from internal storage to a file. The type of the items specified in the NAMELIST statement determines the conversions to be performed.

Syntax

WRITE
  ([UNIT=]*un***,**
  [FMT=]*name*
  [, ERR=*stl***]**
  [, IOSTAT=*ios***])**

UNIT=*un*

un is the external unit identifier. *un* is one of the following:

- An integer expression of length 4 whose value must be zero or positive, or
- An asterisk (*) representing an installation-dependent unit.

un is required in the first form of the WRITE statement and can, optionally, be preceded by UNIT= If UNIT= is omitted, *un* must appear immediately following the WRITE statement. The other specifiers may appear in any order. If UNIT= is included on the WRITE statement, all the specifiers can appear in any order.

In the form of the WRITE where *un* is designated, *un* is installation dependent.

FMT=*name*

name is a NAMELIST name. See "NAMELIST Statement" on page 148.

If FMT= is omitted, the NAMELIST name must appear second. If both UNIT= and FMT= are included on the WRITE statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

Valid WRITE Statements:

```
WRITE (101,DALMATIANS)

WRITE (IN+8, NAMEOUT, IOSTAT=IACT(1), ERR=99999)
```

Invalid WRITE Statements:

```
WRITE (APOLLO,12)                un must appear before name.

WRITE (5,GOLDEN_RINGS) Q1,Q2,Q3  list must not be specified.
```

If the unit specified by *un* is encountered, it must exist and must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

A BACKSPACE or REWIND statement should not be used for a file that is written using NAMELIST. If it is, the results are unpredictable (see "BACKSPACE Statement" on page 57).

Data Transmission: A WRITE statement with NAMELIST starts data transmission from the beginning of a record. The data is taken from each item in the NAMELIST with *name* in the order in which they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item in the NAMELIST name.

After execution of a WRITE statement with NAMELIST, no record exists in the file following the end of the NAMELIST just transmitted.

Data and NAMELIST: The NAMELIST name must appear on the external file.

The number of characters specified by the items in the NAMELIST name and the necessary data separators and identifiers are placed on tne external file.

For information on how to calculate the size of the record on the external file, see Figure 25 on page 230. It shows the width of the written field for any item's data type and length. The size of the record will be the sum of the field widths plus:

- The number of bytes needed for each item's name and an equal sign (these are prefixed to each field), and
- A byte to separate each field

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be written; only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

WRITE Statement—NAMELIST with Internal Files

This statement transfers data from one or more areas of internal storage to another area of internal storage. The receiving area is called an internal file. This statement can be used to convert numeric data to character data. The type of the items specified in an associated NAMELIST list determines the conversions to be performed.

Syntax

```
WRITE
      ( [UNIT=]un,
      [FMT=]name
      [, ERR=stl]
      [, IOSTAT=ios] )
```

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It must be the name of a character array with at least three elements.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not omitted, *un* must appear immediately following the left parenthesis. If UNIT= is included on the WRITE statement, FMT= must be used and all the specifiers can appear in any order.

FMT=*name*

name is a NAMELIST name. See "NAMELIST Statement" on page 148.

If FMT= is omitted, the NAMELIST name must appear second. If both UNIT= and FMT= are included on the WRITE statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error condition is detected.

Valid WRITE Statements:

```
WRITE (77,SUNSET_STRIP)

WRITE (IN+8, NAMOUT, IOSTAT=IACT(1), ERR=99999)
```

Data Transmission: A WRITE statement with NAMELIST starts data transmission from the beginning of the internal file. The data is taken from each item in the list associated with the NAMELIST name, in the order in which the items are specified, and transmitted to the internal file. Data transmission stops when data has been transferred from every item in the list.

Data and NAMELIST: The NAMELIST name must appear in the internal file.

The number of characters specified by the items in the NAMELIST name and the necessary data separators and identifiers are placed in the internal file.

For information on how to calculate the size of the internal file, see Figure 25 on page 230. It shows the width of the written field for any item's data type and length. The size of the internal file will be the sum of the field widths plus:

- The number of bytes needed for each item's name and an equal sign (these are prefixed to each field), and
- A byte to separate each field.

Example:

```
NAMELIST  /NL1/I,J,C
CHARACTER*40 CHAR(3)
CHARACTER*5  C
INTEGER*2    I,J
I=12046
J=12047
C='BACON'
WRITE(CHAR,NL1)
```

After execution of the WRITE statement:

```
                  Position 2
                  v
CHAR(1) contains  &NL1
CHAR(2) contains  I= 12046,J= 12047,C='BACON'
CHAR(3) contains  &END
```

WRITE Statement—Unformatted with Direct Access

This statement transfers data without conversion from internal storage to a file. The data must be sent to an external file that is connected with direct access to a unit. (See "OPEN Statement" on page 151.)

Syntax

WRITE
([UNIT=]*un***,**
REC=*rec*
[, ERR=*stl***]**
[, IOSTAT=*ios***]**
[, NUM=*n***])**
[*list***]**

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the WRITE statement, all the specifiers can appear in any order.

REC=*rec*

rec is an integer expression. It represents the relative position of a record within the file associated with *un*.

Its value after conversion to integer, if necessary, must be greater than zero. The internal record number of the first record is 1. The INQUIRE statement can be used to determine the record number.

This specifier is required. (Note that, if *list* is omitted, a blank record is transmitted to the output device.)

ERR=*stl*
stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

NUM=*n*
n is an integer variable or an integer array element of length 4.

If NUM=*n* is specified, the variable or array element *n* is assigned a value representing the number of bytes of data transmitted to the elements specified by the I/O list.

Coding the NUM specifier suppresses the indication of an error that would occur if the number of bytes represented by the I/O list is greater than the number of bytes that can be written into the record. In this case, *n* is set to a value that is the maximum length record that can be written. Data from remaining I/O list items is not written into subsequent records.

list
is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 81. If *list* is omitted, a blank record is transmitted to the output device.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (IOSTAT=IOS, ERR=99999, REC=IN-3, UNIT=IN+6)

WRITE (IOSTAT=IACT(1), REC=2*IN-7, UNIT=2*IN+1) EXPECT(3)

WRITE (REC=1, UNIT=11) EXPECT(1)
```

If the unit specified by *un* is encountered, it must exist and the file must be connected for direct access.

Data Transmission: A WRITE statement without conversion starts data transmission at the record specified by *rec*. The data is taken from the items of the list in the order in which they are specified and transmitted onto the record *rec* of the external file. Data transmission stops when data has been transferred from every item of the list.

Data and I/O List: The length of every FORTRAN record is designated by the RECL specifier of the OPEN statement. If the length of the record *rec* is *greater* than the total amount of data transmitted from the items of the list, the remainder of the record is filled with zeros. If the length of the record *rec* is *smaller* than the total amount of data transmitted from the items of the list, as much data as can fit in the record is written, and an error is detected unless the NUM specifier is given.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be written; only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor transfer to be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

WRITE Statement—Unformatted with Keyed Access

This statement transfers data without conversion from internal storage to a file. The data must be sent to an external file that is connected with keyed access to a unit (see "OPEN Statement" on page 151).

Syntax

```
WRITE
      ( [UNIT=]un
      [, ERR=stl]
      [, IOSTAT=ios]
      [, DUPKEY=stl]
      [, NUM=n] )
      list
```

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the WRITE statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

DUPKEY=*stl*

stl is the statement label of a statement to which control is passed when a duplicate-key condition occurs. For an explanation of this condition, see "Duplicate Key," below.

NUM=*n*

n is an integer variable or an integer array element of length 4.

If NUM=*n* is specified, the variable or array element *n* is assigned a value representing the number of bytes of data transmitted to the elements specified by the I/O list.

Coding the NUM specifier suppresses the indication of an error that would occur if the number of bytes represented by the I/O list is greater than the number of bytes that can be written into the record. In this case, *n* is set to a value that is the maximum length record that can be written. Data from remaining I/O list items is not written into subsequent records.

list

is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 81.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (12) GG,HH,II

WRITE (12,DUPKEY=55) DD,EE,FF
```

If this WRITE statement is encountered, the unit specified must exist and the file must have been connected for keyed access by means of an OPEN statement. The ACTION specifier of that OPEN statement must have specified the value 'READWRITE' or 'WRITE'. If the file is not so connected, an error is detected.

Data Transmission: If the WRITE statement was issued for a file connected by an OPEN statement with an ACTION specifier of 'WRITE', data transmission begins at the beginning of a new record. The new record will follow, in order of key value, the last record written. If the file was connected by an OPEN statement with an ACTION specifier of 'READWRITE', data transmission also begins at the beginning of a new record. In this case, however, the new record will be inserted following the record with a lower key value and preceding the record with a higher key value. If the new record has a key that is the same as a key already in the file, the new record is added following the last record with the same key. The data is taken from the items in the list in the order they are specified; the data is transmitted onto a single record of the file. Data transmission stops when data has been transferred from every item in the list.

Data and I/O List: The amount of data specified by the items of the list defines the length of the record to be written. A single WRITE statement creates only one record. The record must be long enough to include all the keys that are defined for the file.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be written; only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

Duplicate Key: Control is transferred to the statement specified by DUPKEY when a duplicate-key condition occurs; namely:

- The file is connected by an OPEN statement with an ACTION specifier of 'READWRITE', or when ACTION='WRITE', and
- An attempt was made to write a record with a key whose values must be unique, and
- The key value would have duplicated one that already exists for the same key in another record.

If IOSTAT=*ios* is specified, a positive integer value is assigned to *ios* when the duplicate-key condition is detected. If ERR is specified but DUPKEY is not, control passes to the statement specified by ERR when the duplicate-key condition is detected. If neither DUPKEY nor ERR was given, an error is detected.

Examples:

```
WRITE (UNIT=10) AA, BB, CC
WRITE (10,DUPKEY=77) AA, BB, CC
WRITE (10,NUM=LENG) AA, BB, CC
```

WRITE Statement—Unformatted with Sequential Access

This statement transfers data without conversion from internal storage to a file. The data must be sent to an external file that is connected with sequential access to a unit (see "Input/Output Semantics" on page 46).

Syntax

WRITE
([UNIT=]*un*
[, ERR=*stl***]**
[, IOSTAT=*ios***]**
[, NUM=*n***])**
[*list***]**

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may appear in any order. If UNIT= is included on the WRITE statement, all the specifiers can appear in any order.

ERR=*stl*

stl is the statement label of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error condition is detected. For VSAM files, return and reason codes are placed in *ios*.

NUM=*n*

n is an integer variable or an integer array element of length 4.

If NUM=*n* is specified, the variable or array element *n* is assigned a value representing the number of bytes of data transmitted to the elements specified by the I/O list. Coding the NUM specifier suppresses the indication of an error that would occur if the number of bytes represented by the I/O list is greater than the number of bytes that can be written into the record. In this case, *n* is set to a value which is the maximum length record that can be written. Data from remaining I/O list items is not written into subsequent records.

list

is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 81.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Valid WRITE Statements:

```
WRITE (16) G,N,P

WRITE (UNIT=16) TAXES(1)

WRITE(5) EXPECT(4)
```

Invalid WRITE Statement:

```
WRITE 5, EXPECT(4)      un must be in parentheses.
```

Data Transmission: A WRITE statement without conversion starts data transmission at the beginning of a record. The data is taken from the items of the list in the order in which they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item of the list.

After execution of a sequential WRITE statement, no record exists in the file following the last record transferred by that statement.

The WRITE statement writes over an end of file and extends the external file. An END FILE, BACKSPACE, CLOSE, or REWIND statement will then reinstate the end of file.

Data and I/O List: The amount of character data specified by the items of the list defines the length of the FORTRAN record (also called a logical record). A single WRITE statement creates only one FORTRAN record.

VS FORTRAN Version 2 Programming Guide describes how to associate FORTRAN records (that is, logical records) and records in an external device.

If a transmission error is detected, control is transferred to the statement specified by ERR. No indication is given of which record or records could not be written; only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.)

Chapter 5. Intrinsic Functions

Intrinsic functions are procedures supplied in VS FORTRAN Version 2 for standard mathematical computations, character manipulations, and bit manipulations. A function is invoked by including its name in an arithmetic or character expression accompanied by one or more arguments. The compiler recognizes the function by its name, checks the syntax of the arguments, and generates code that performs the desired function.

The general format for referring to an intrinsic function is

```
name (arg1[,arg2...])
```

where *name* is the function name and *arg1*, *arg2*, ... are the actual arguments.

For example, the source statement

```
SINRAD=SIN(RADIAN)
```

causes the sine function (SIN) to be invoked. The value of the argument RADIAN is given to the sine function, which computes the sine of that value. The result is stored in the variable SINRAD.

Nearly all the mathematical functions have both generic and specific names. Use of the generic name simplifies the referencing of the functions, because the same name may be used for the entire range of argument types permitted. The appropriate specific entry name is chosen (by the compiler) when the generic name is used, based on the type of argument(s) presented.

Figure 33 on page 259 lists all the generic function names and gives the valid range of argument types and function values for them.

The intrinsic functions provided by VS FORTRAN Version 2 are described in detail in the following figures, grouped by function:

Function	Figure
Logarithmic and Exponential	Figure 26 on page 246
Trigonometric	Figure 27 on page 248
Hyperbolic	Figure 28 on page 250
Miscellaneous Mathematical	Figure 29 on page 250
Conversion and Maximum/Minimum	Figure 30 on page 253
Character Manipulation	Figure 31 on page 256
Bit Manipulation	Figure 32 on page 257

All the specific function names listed in Figure 26 through Figure 29, and in Figure 32, can be passed as actual arguments. None of the function names listed in Figure 30 or Figure 31 (except for LEN and INDEX) can be passed as actual arguments. (An INTRINSIC statement for a specific function name must appear in any program unit that passes the name as an actual argument.)

References to the functions are either resolved from the library or inserted in the object module. That is, the code generated by VS FORTRAN Version 2 for the reference contains either instructions to link to the function in the library (out-of-line) or instructions to perform the function directly (in-line). Notes with the figures state whether the functions are performed in-line or out-of-line.

For a small subset of the mathematical functions, alternative functions are available that under certain conditions provide greater accuracy and faster computation. These functions are identified in footnotes to the figures. For more information, see Chapter 6, "Mathematical, Character, and Bit Routines" on page 263.

The following information is provided for each entry name in Figure 26 on page 246 through Figure 32 on page 257:

General Function: This column states the nature of the computation performed by the function.

Generic Name: This column gives the generic name of the function (if any).

Entry Name: This column gives the specific entry names of the function. A function may have more than one entry name; the particular entry name used depends on the computation to be performed. For example, the sine and cosine function has two entry names: SIN and COS. If the sine is to be computed, entry name SIN is used; if the cosine is to be computed, entry name COS is used.

Definition: This column gives a mathematical equation that represents the computation. An alternative equation is given in those cases in which there is another way of representing the computation in mathematical notation. For example, the square root can be represented either as:

$y=\sqrt{x}$ or $y=x^{1/2}$

Argument Number: This column states how many arguments the programmer must supply.

Argument Type: This column describes the type and length of each argument. INTEGER, REAL, COMPLEX, LOGICAL, and character represent the type; the notations *1, *4, *8, *16, *32, and *n represent the size of the argument in number of storage locations. (The notation *n describes character data.)

Argument Range: This column gives the valid range for arguments. If an argument is not within the range, an error message is issued (see Error Code column).

Function Value Type and Range: This column describes the type and range of the function value returned by the subprogram. Type notation used is the same as that for the argument type. The range symbol is

$\gamma = 16^{63}\ (1-16^{-6})$

for regular precision routines;

$\gamma = 16^{63}\ (1-16^{-14})$

for double-precision; and

$$\gamma = 16^{63}\ (1-16^{-28})$$

for extended precision.

Error Code: This column gives the number of the message issued when an error occurs. Appendix D, "Library Procedures and Messages" on page 375 contains descriptions of the error messages.

Throughout these figures, the following approximate values are represented by

$(2^{18} \bullet \pi)$ and $(2^{50} \bullet \pi)$:

$$(2^{18} \bullet \pi) = .82354966458264280 + 06$$
$$(2^{50} \bullet \pi) = .35371188760142200 + 16$$

General Function[6]	Entry Name	Definition	Number	Argument(s) Type[1]	Range	Function Value Type and Range[1],[4]	Error Code
Common and natural logarithm	ALOG	$y = \log_e x$ or $y = \ln x$ Note 5	1	REAL*4	$x > 0$	REAL*4 $y \geq -180.218$ $y \leq 174.673$	253
	ALOG10	$y = \log_{10} x$ Note 5	1	REAL*4	$x > 0$	REAL*4 $y \geq -78.268$ $y \leq 75.859$	253
	DLOG	$y = \log_e x$ or $y = \ln x$ Note 5	1	REAL*8	$x > 0$	REAL*8 $y \geq -180.218$ $y \leq 174.673$	263
	DLOG10	$y = \log_{10} x$ Note 5	1	REAL*8	$x > 0$	REAL*8 $y \geq -78.268$ $y \leq 75.859$	263
	CLOG	$y = PV \log_e (z)$ Note 2	1	COMPLEX*8	$z \neq 0 + 0i$	COMPLEX*8 $y_1 \geq -180.218$ $y_1 \leq 175.021$ $-\pi \leq y_2 \leq \pi$	273
	CDLOG	$y = PV \log_e (z)$ Note 2	1	COMPLEX*16	$z \neq 0 + 0i$	COMPLEX*16 $y_1 \geq -180.218$ $y_1 \leq 175.021$ $-\pi \leq y_2 \leq \pi$	283
	QLOG	$y = \log_e x$ or $y = \ln x$	1	REAL*16	$x > 0$	REAL*16 $y \geq -180.218$ $y \leq 174.673$	293
	QLOG10	$y = \log_{10} x$	1	REAL*16	$x > 0$	REAL*16 $y \geq -78.268$ $y \leq 175.859$	293
	CQLOG	$y = PV \log_e (z)$ Note 2	1	COMPLEX*32	$z \neq 0 + 0i$	COMPLEX*32 $y_1 \geq -180.218$ $y_1 \leq 175.021$ $-\pi \leq y_2 \leq \pi$	278
Exponential	EXP	$y = e^x$ Note 5	1	REAL*4	$x \leq 174.673$	REAL*4 $0 \leq y \leq \gamma$	252
	DEXP	$y = e^x$ Note 5	1	REAL*8	$x \leq 174.673$	REAL*8 $0 \leq y \leq \gamma$	262
	CEXP	$y = e^z$ Note 3	1	COMPLEX*8	$x_1 \leq 174.673$ $\lvert x_2 \rvert < (2^{18} \bullet \pi)$	COMPLEX*8 $-\gamma \leq y_1, y_2 \leq \gamma$	271, 272
	CDEXP	$y = e^z$ Note 3	1	COMPLEX*16	$x_1 \leq 174.673$ $\lvert x_2 \rvert < (2^{50} \bullet \pi)$	COMPLEX*16 $-\gamma \leq y_1, y_2 \leq \gamma$	281, 282
	QEXP	$y = e^x$	1	REAL*16	$x \geq -180.218$ $x \leq 174.673$	REAL*16 $0 \leq y \leq \gamma$	292
	CQEXP	$y = e^z$ Note 3	1	COMPLEX*32	$x_1 \leq 174.673$ $x_2 \leq 2^{100}$	COMPLEX*32 $-\gamma \leq y_1, y_2 \leq \gamma$	276, 277

Figure 26. Logarithmic and Exponential Functions

Notes to Figure 26:

[1] REAL*4, REAL*8, and REAL*16 arguments correspond to REAL, DOUBLE PRECISION, and EXTENDED PRECISION arguments, respectively, in VS FORTRAN Version 2.

[2] *PV* = principal value. The answer given ($y_1 + y_2 i$) is that one whose imaginary part (y_2) lies between $- \pi$ and $+ \pi$. More specifically: $- \pi < y_x \leq \pi$, unless $x_1 < 0$ and $x_2 = -0$, in which case, $y_2 = - \pi$.

[3] *z* is a COMPLEX number of the form $x_1 + x_2 i$.

[4] $\gamma = 16^{63}(1 - 16^{-6})$ for regular precision routines, $16^{63}(1 - 16^{-14})$ for double precision routines, and $16^{63}(1 - 16^{-28})$ for extended precision.

[5] Available also in the alternate mathematical library.

[6] All functions are generated as out-of-line library calls.

General Function[6]	Entry Name	Definition	Number	Argument(s) Type[1]	Range	Function Value Type and Range[1], [5]	Error Code
Arcsine and arccosine	ASIN	$y = \arcsin(x)$ Note 7	1	REAL*4	$\lvert x \rvert \le 1$	REAL*4 (in radians) $-\frac{\pi}{2} \le y \le \frac{\pi}{2}$	257
	ACOS	$y = \arccos(x)$ Note 7	1	REAL*4	$\lvert x \rvert \le 1$	REAL*4 (in radians) $0 \le y \le \pi$	257
	DASIN	$y = \arcsin(x)$ Note 7	1	REAL*8	$\lvert x \rvert \le 1$	REAL*8 (in radians) $-\frac{\pi}{2} \le y \le \frac{\pi}{2}$	267
	DACOS	$y = \arccos(x)$ Note 7	1	REAL*8	$\lvert x \rvert \le 1$	REAL *8 (in radians) $0 \le y \le \pi$	267
	QARSIN	$y = \arcsin(x)$	1	REAL*16	$\lvert x \rvert \le 1$	REAL*16 $-\frac{\pi}{2} \le y \le \frac{\pi}{2}$	297
	QARCOS	$y = \text{arcos}(x)$	1	REAL*16	$\lvert x \rvert \le 1$	REAL*16 $0 \le y \le \pi$	297
Arctangent	ATAN	$y = \arctan(x)$ Note 7	1	REAL*4	any REAL argument	REAL*4 (in radians) $-\frac{\pi}{2} \le y \le \frac{\pi}{2}$	None
	ATAN2	$y = \arctan\left(\frac{x_1}{x_2}\right)$ Note 7	2	REAL*4	any REAL arguments (except 0, 0)	REAL*4 (in radians) $-\pi < y \le \pi$	255
	DATAN	$y = \arctan(x)$ Note 7	1	REAL*8	any REAL argument	REAL*8 (in radians) $-\frac{\pi}{2} \le y \le \frac{\pi}{2}$	None
	DATAN2	$y = \arctan\left(\frac{x_1}{x_2}\right)$ Note 7	2	REAL*8	any REAL arguments (except 0, 0)	REAL*8 (in radians) $-\pi < y \le \pi$	265
	QATAN	$y = \arctan(x)$	1	REAL*16	any REAL argument	REAL*16 (in radians) $-\frac{\pi}{2} \le y \le \frac{\pi}{2}$	None
	QATAN2	$y = \arctan\left(\frac{x_1}{x_2}\right)$	2	REAL*16	any REAL arguments (except 0, 0)	REAL*16 (in radians) $-\pi < y \le \pi$	295
Sine and cosine	SIN	$y = \sin(x)$ Note 7	1	REAL*4 (in radians)	$\lvert x \rvert < (2^{18} \bullet \pi)$	REAL*4 $-1 \le y \le 1$	254
	COS	$y = \cos(x)$ Note 7	1	REAL*4 (in radians)	$\lvert x \rvert < (2^{18} \bullet \pi)$	REAL*4 $-1 \le y \le 1$	254
	DSIN	$y = \sin(x)$ Note 7	1	REAL*8 (in radians)	$\lvert x \rvert < (2^{50} \bullet \pi)$	REAL*8 $-1 \le y \le 1$	264
	DCOS	$y = \cos(x)$ Note 7	1	REAL*8 (in radians)	$\lvert x \rvert < (2^{50} \bullet \pi)$	REAL*8 $-1 \le y \le 1$	264
	CSIN	$y = \sin(z)$ Note 2	1	COMPLEX*8 (in radians)	$\lvert x_1 \rvert < (2^{18} \bullet \pi)$ $\lvert x_2 \rvert \le 174.673$	COMPLEX*8 $-\gamma \le y_1, y_2 \le \gamma$	274, 275
	CCOS	$y = \cos(z)$ Note 2	1	COMPLEX*8 (in radians)	$\lvert x_1 \rvert < (2^{18} \bullet \pi)$ $\lvert x_2 \rvert \le 174.673$	COMPLEX*8 $-\gamma \le y_1, y_2 \le \gamma$	274, 275
	CDSIN	$y = \sin(z)$ Note 2	1	COMPLEX*16 (in radians)	$\lvert x_1 \rvert < (2^{50} \bullet \pi)$ $\lvert x_2 \rvert \le 174.673$	COMPLEX*16 $-\gamma \le y_1, y_2 \le \gamma$	284, 285
	CDCOS	$y = \cos(z)$ Note 2	1	COMPLEX*16 (in radians)	$\lvert x_1 \rvert < (2^{50} \bullet \pi)$ $\lvert x_2 \rvert \le 174.673$	COMPLEX*16 $-\gamma \le y_1, y_2 \le \gamma$	284, 285
	QSIN	$y = \sin(x)$	1	REAL*16 (in radians)	$\lvert x \rvert < 2^{100}$	REAL*16 $-1 \le y \le 1$	294

Figure 27 (Part 1 of 2). Trigonometric Functions

General Function[6]	Entry Name	Definition	Number	Argument(s) Type[1]	Range	Function Value Type and Range[1], [5]	Error Code
	QCOS	$y = \cos(x)$	1	REAL*16 (in radians)	$\vert x \vert < 2^{100}$	REAL*16 $-1 \le y \le 1$	294
	CQSIN	$y = \sin(z)$ Note 2	1	COMPLEX*32 (in radians)	$\vert x_1 \vert < 2^{100}$ $\vert x_2 \vert \le 174.673$	COMPLEX*32 $-\gamma \le y_1, y_2 \le \gamma$	279, 280
	CQCOS	$y = \cos(z)$ Note 2	1	COMPLEX*32 (in radians)	$\vert x_1 \vert < 2^{100}$ $\vert x_2 \vert \le 174.673$	COMPLEX*32 $-\gamma \le y_1, y_2 \le \gamma$	279, 280
Tangent and cotangent	TAN	$y = \tan(x)$ Note 7	1	REAL*4 (in radians)	$\vert x \vert < (2^{18} \bullet \pi)$ Note 4	REAL*4 $-\gamma \le y \le \gamma$	258
	COTAN	$y = \cot(x)$ Note 7	1	REAL*4 (in radians)	$\vert x \vert < (2^{18} \bullet \pi)$ Note 4	REAL*4 $-\gamma \le y \le \gamma$	258, 259
	DTAN	$y = \tan(x)$ Note 7	1	REAL*8 (in radians)	$\vert x \vert < (2^{50} \bullet \pi)$ Note 4	REAL*8 $-\gamma \le y \le \gamma$	268
	DCOTAN	$y = \cot(x)$ Note 7	1	REAL*8 (in radians)	$\vert x \vert < (2^{50} \bullet \pi)$ Note 4	REAL*8 $-\gamma \le y \le \gamma$	268, 269
	QTAN	$y = \tan(x)$	1	REAL*16 (in radians)	$\vert x \vert < 2^{100}$ Note 3	REAL*16 $-\gamma \le y \le \gamma$	298, 299
	QCOTAN	$y = \cot(x)$	1	REAL*16 (in radians)	$\vert x \vert < 2^{100}$ $\vert x \vert > 16^{-63}$ Note 3	REAL*16 $-\gamma \le y \le \gamma$	298, 299

Figure 27 (Part 2 of 2). Trigonometric Functions

Notes to Figure 27:

[1] REAL*4, REAL*8, and REAL*16 correspond to REAL, DOUBLE PRECISION, and EXTENDED PRECISION arguments, respectively, in VS FORTRAN Version 2.

[2] z is a complex number of the form $x_1 + x_2 i$.

[3] x may not be such that one can find a singularity within 8 units of the last digit value of the floating-point representation of x. Singularities are $\pm (2n + 1)\pi/2$, $n = 0, 1, 2, \ldots$ for tangent, and $\pm n\pi$, $n = 0, 1, 2, \ldots$ for cotangent.

[4] The argument for the cotangent functions may not approach a multiple of π; the argument for the tangent functions may not approach an odd multiple of $\pi/2$.

[5] $\gamma = 16^{63}(1 - 16^{-6})$ for regular precision routines, $16^{63}(1 - 16^{-14})$ for double-precision routines and $16^{63}(1 - 16^{-28})$ for extended precision.

[6] All functions are generated as out-of-line library calls.

[7] Available also in the alternate mathematical library.

General Function[3]	Entry Name	Definition	Number	Argument(s) Type[1]	Range	Function Value Type and Range[1], [2]	Error Code
Hyperbolic sine and cosine	SINH	$y = \frac{e^x - e^{-x}}{2}$ Note 4	1	REAL*4	$\lvert x \rvert < 175.366$	REAL*4 $-\gamma \le y \le \gamma$	256
	COSH	$y = \frac{e^x + e^{-x}}{2}$ Note 4	1	REAL*4	$\lvert x \rvert < 175.366$	REAL*4 $1 \le y \le \gamma$	256
	DSINH	$y = \frac{e^x - e^{-x}}{2}$	1	REAL*8	$\lvert x \rvert < 175.366$	REAL*8 $-\gamma \le y \le \gamma$	266
	DCOSH	$y = \frac{e^x + e^{-x}}{2}$	1	REAL*8	$\lvert x \rvert < 175.366$	REAL*8 $1 \le y \le \gamma$	266
	QSINH	$y = \frac{e^x - e^{-x}}{2}$	1	REAL*16	$\lvert x \rvert \le 175.366$	REAL*16 $-\gamma \le y \le \gamma$	296
	QCOSH	$y = \frac{e^x + e^{-x}}{2}$	1	REAL*16	$\lvert x \rvert \le 175.366$	REAL*16 $1 \le y \le \gamma$	296
Hyperbolic tangent	TANH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ Note 4	1	REAL*4	any REAL argument	REAL*4 $-1 \le y \le 1$	None
	DTANH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1	REAL*8	any REAL argument	REAL*8 $-1 \le y \le 1$	None
	QTANH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1	REAL*16	any REAL argument	REAL*16 $-1 \le y \le 1$	None

Figure 28. Hyperbolic Functions

Notes to Figure 28:

[1] REAL*4, REAL*8, and REAL*16 arguments correspond to REAL, DOUBLE PRECISION, and EXTENDED PRECISION arguments, respectively, in VS FORTRAN Version 2.

[2] $\gamma = 16^{63}(1 - 16^{-6})$ for regular precision routines, $16^{63}(1 - 16^{-16})$ for double-precision routines, and $16^{63}(1 - 16^{-28})$ for extended precision.

[3] All functions are generated as out-of-line library calls.

[4] Available also in the alternate mathematical library.

General Function	Entry Name	Definition	Number	Argument(s) Type[1]	Range	Function Value Type and Range[1], [5]	Error Code
Absolute value	IABS	$y = \lvert x \rvert$	1	INTEGER*4	any INTEGER argument	INTEGER*4 $0 \le y \le \gamma$	None Note 9
	ABS	$y = \lvert x \rvert$	1	REAL*4	any REAL argument	REAL*4 $0 \le y \le \gamma$	None Note 9
	DABS	$y = \lvert x \rvert$	1	REAL*8	any REAL argument	REAL*8 $0 \le y \le \gamma$	None Note 9
	QABS	$y = \lvert x \rvert$	1	REAL*16	any REAL argument	REAL*16 $0 \le y \le \gamma$	None Note 9
	CABS	$y = \lvert z \rvert = (x_1{}^2 + x_2{}^2)^{1/2}$ Note 11	1	COMPLEX*8	any COMPLEX argument Note 2	REAL*4 $0 \le y_1 \le \gamma$ $y_2 = 0$	None Note 10
	CDABS	$y = \lvert z \rvert = (x_1{}^2 + x_2{}^2)^{1/2}$ Note 11	1	COMPLEX*16	any COMPLEX argument Note 2	REAL*8 $0 \le y_1 \le \gamma$ $y_2 = 0$	None Note 10

Figure 29 (Part 1 of 4). Miscellaneous Mathematical Functions

General Function	Entry Name	Definition	Number	Argument(s) Type[1]	Range	Function Value Type and Range[1], [5]	Error Code
	CQABS	$y = \lvert z \rvert = (x_1{}^2 + x_2{}^2)^{1/2}$	1	COMPLEX*32	any COMPLEX argument Note 2	REAL*16 $0 \leq y_1 \leq \gamma$ $y_2 = 0$	None Note 10
Error function	ERF	$y = \frac{2}{\sqrt{\pi}} \int_0^x e^{-u^2} du$	1	REAL*4	any REAL argument	REAL*4 $-1 \leq y \leq 1$	None Note 10
	ERFC	$y = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-u^2} du$ $y = 1 - \text{ERF}(x)$	1	REAL*4	any REAL argument	REAL*4 $0 \leq y \leq 2$	None Note 10
	DERF	$y = \frac{2}{\sqrt{\pi}} \int_0^x e^{-u^2} du$	1	REAL*8	any REAL argument	REAL*8 $-1 \leq y \leq 1$	None Note 10
	DERFC	$y = \frac{2}{\sqrt{\pi}} \int_0^\infty e^{-u^2} du$ $y = 1 - \text{ERF}(x)$	1	REAL*8	any REAL argument	REAL*8 $0 \leq y \leq 2$	None Note 10
	QERF	$y = \frac{2}{\sqrt{\pi}} \int_0^x e^{-u^2} du$	1	REAL*16	any REAL argument	REAL*16 $-1 \leq y \leq 1$	None Note 10
	QERFC	$y = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-u^2} du$ $y = 1 - \text{ERF}(x)$	1	REAL*16	any REAL argument	REAL*16 $0 \leq y \leq 2$	None Note 10
Gamma and log gamma	GAMMA	$y = \int_0^\infty u^{x-1} e^{-u} du$	1	REAL*4	$x > 2^{-252}$ and $x < 57.5744$	REAL*4 $0.88560 \leq y \leq \gamma$	290 Note 10
	ALGAMA	$y = \log_e \Gamma(x)$ or $y = \log_e \int_0^\infty u^{x-1} e^{-u} du$	1	REAL*4	$x > 0$ and $x < 4.2913 \cdot 10^{73}$	REAL*4 $-0.12149 \leq y \leq \gamma$	291 Note 10
	DGAMMA	$y = \int_0^\infty u^{x-1} e^{-u} du$	1	REAL*8	$x > 2^{-252}$ and $x < 57.5744$	REAL*8 $0.88560 \leq y \leq \gamma$	300 Note 10
	DLGAMA	$y = \log_e \Gamma(x)$ or $y = \log_e \int_0^\infty u^{x-1} e^{-u} du$	1	REAL*8	$x > 0$ and $x \leq 4.2913 \cdot 10^{73}$	REAL*8 $-0.12149 \leq y \leq \gamma$	301 Note 10
Square root	SQRT	$y = \sqrt{x}$ or $y = x^{1/2}$ Note 11	1	REAL*4	$x \geq 0$	REAL*4 $0 \leq y \leq \gamma^{1/2}$	251 Note 10
	DSQRT	$y = \sqrt{x}$ or $y = x^{1/2}$ Note 11	1	REAL*8	$x \geq 0$	REAL*8 $0 \leq y \leq \gamma^{1/2}$	261 Note 10
	CSQRT	$y = \sqrt{z}$ or $y = z^{1/2}$ Note 7	1	COMPLEX*8	any COMPLEX argument	COMPLEX*8 $0 \leq y_1 \leq 1.0987(\gamma^{1/2})$ $\lvert y_2 \rvert \leq 1.0987(\gamma^{1/2})$	None Note 10
	CDSQRT	$y = \sqrt{z}$ or $y = z^{1/2}$ Note 7	1	COMPLEX*16	any COMPLEX argument	COMPLEX*16 $0 \leq y_1 \leq 1.0987(\gamma^{1/2})$ $\lvert y_2 \rvert \leq 1.0987(\gamma^{1/2})$	None Note 10
	QSQRT	$y = \sqrt{x}$ or $y = x^{1/2}$	1	REAL*16	$x \geq 0$	REAL*16 $0 \leq y \leq y^{1/2}$	289 Note 10
	CQSQRT	$y = \sqrt{z}$ or $y = z^{1/2}$ Note 7	1	COMPLEX*32	any COMPLEX argument	COMPLEX*32 $0 \leq y_1 \leq 1.0987(\gamma^{1/2})$ $y_2 \leq 1.0987(\gamma^{1/2})$	None Note 10
Modular arithmetic	MOD	$y = x_1$ (modulo x_2) Note 3	2	INTEGER	$x_2 \neq 0$ Note 4	INTEGER*4	None Note 9
	AMOD		2	REAL*4	$x_2 \neq 0$ Note 4	REAL*4	None Note 9
	DMOD		2	REAL*8	$x_2 \neq 0$ Note 4	REAL*8	None Note 9

Figure 29 (Part 2 of 4). Miscellaneous Mathematical Functions

General Function	Entry Name	Definition	Number	Argument(s) Type[1]	Range	Function Value Type and Range[1], [5]	Error Code
	QMOD		2	REAL*16	$x_2 \neq 0$ Note 4	REAL*16	None Note 9
Truncation	AINT	y = (sign of x) • n where $n = [\,|x|\,]$ Note 6	1	REAL*4	any REAL argument	REAL*4	None Note 9
	DINT		1	REAL*8	any REAL argument	REAL*8	None Note 9
	QINT		1	REAL*16	any REAL argument	REAL*16	None Note 9
Obtain imaginary part of a complex argument	AIMAG		1	COMPLEX*8	any COMPLEX argument	REAL*4	None Note 9
	DIMAG		1	COMPLEX*16	any COMPLEX argument	REAL*8	None Note 9
	QIMAG		1	COMPLEX*32	any COMPLEX argument	REAL*16	None Note 9
Obtain conjugate of a complex argument	CONJG	$y = x_1 - x_2 i$ for argument $= x_1 + x_2 i$	1	COMPLEX*8	any COMPLEX argument	COMPLEX*8	None Note 9
	DCONJG		1	COMPLEX*16	any COMPLEX argument	COMPLEX*16	None Note 9
	QCONJG		1	COMPLEX*32	any COMPLEX argument	COMPLEX*32	None Note 9
Nearest whole number	ANINT	y = (sign of x) • v where $v = [\,|x + .5|\,]$ if $x \geq 0$ or	1	REAL*4	any REAL argument	REAL*4	None Note 9
	DNINT	$v = [\,|x - .5|\,]$ if $x < 0$. Note 6	1	REAL*8	any REAL argument	REAL*8	None Note 9
Nearest integer	NINT	y = (sign of x) • n where $n = [\,|x + .5|\,]$ if $x \geq 0$ or	1	REAL*4	any REAL argument	INTEGER*4	None Note 9
	IDNINT	$n = [\,|x - .5|\,]$ if $x < 0$. Note 8	1	REAL*8	any REAL*8 argument	INTEGER*4	None Note 9
Positive difference	IDIM	$y = x_1 - x_2$ if $x_1 > x_2$ $y = 0$ if $x_1 \leq x_2$	2	INTEGER*4	any INTEGER argument	INTEGER*4	None Note 9
	DIM		2	REAL*4	any REAL argument	REAL*4	None Note 9
	DDIM		2	REAL*8	any REAL argument	REAL*8	None Note 9
	QDIM		2	REAL*16	any REAL argument	REAL*16	None Note 9
Transfer of sign	ISIGN	$y = |x_1|$ if $x_2 \geq 0$ $y = -|x_1|$ if $x_2 < 0$	2	INTEGER*4	any INTEGER argument	INTEGER*4	None Note 9

Figure 29 (Part 3 of 4). Miscellaneous Mathematical Functions

General Function	Entry Name	Definition	Number	Argument(s) Type[1]	Range	Function Value Type and Range[1], [5]	Error Code
	SIGN		2	REAL*4	any REAL argument	REAL*4	None Note 9
	DSIGN		2	REAL*8	any REAL argument	REAL*8	None Note 9
	QSIGN		2	REAL*16	any REAL argument	REAL*16	Note 9
Double precision product	DPROD	$y = x_1 * x_2$	2	REAL*4	any REAL argument	REAL*8	None Note 9

Figure 29 (Part 4 of 4). Miscellaneous Mathematical Functions

Notes to Figure 29:

[1] REAL*4, REAL*8, and REAL*16 arguments correspond to REAL, DOUBLE PRECISION, and EXTENDED PRECISION arguments, respectively, in VS FORTRAN Version 2.

[2] Floating-point overflow can occur.

[3] The expression x_1(modulo x_2) is defined as $x_1 - [x_1/x_2] \bullet x_2$, where the brackets indicate that an integer is used. The largest integer whose magnitude does not exceed the magnitude of x_1/x_2 is used. The sign of the integer is the same as the sign of x_1x_2.

[4] If $x_2 = 0$, the modulus function is mathematically undefined. In addition, a divide exception is recognized and an interruption occurs.

[5] $\gamma = 16^{63}(1 - 16^{-6})$ for regular precision routines, $16^{63}(1 - 16^{-14})$ for double-precision routines, and $16^{63}(1 - 16^{-28})$ for extended precision routines.

[6] $[\,|x|\,]$ is such that $v = |m|$, where m is the greatest integer satisfying the relationship $|m| \leq |x|$, and the resulting v is expressed as a real value.

[7] z is a complex-number of the form $x_1 + x_2i$.

[8] $[\,|x|\,]$ is such that $n = |m|$, where m is the greatest integer satisfying the relationship $|m| \leq |x|$.

[9] This function is generated in-line.

[10] This function is generated as an out-of-line library call.

[11] Available also in the alternate mathematical library.

General Function	Generic Name	Entry Name Name[5]	Definition	Number	Argument(s) Type	Range	Function Value Type and Range
Conversion to integer	INT	Note 1	$y =$ (sign of x) $\bullet\, n$ where n is the largest integer $\leq |x|$	1	INTEGER*4	any INTEGER argument	INTEGER*4
		INT Note 2		1	REAL*4	any REAL argument	INTEGER*4
		IDINT		1	REAL*8	any REAL argument	INTEGER*4
		IQINT		1	REAL*16	any REAL argument	INTEGER*4
		Note 1	for $z = x_1 + x_2i$, $y = \mathrm{INT}(x_1)$	1	COMPLEX*8	any COMPLEX argument	INTEGER*4

Figure 30 (Part 1 of 3). Conversion and Maximum/Minimum Functions

General Function	Generic Name	Entry Name	Definition	Number	Argument(s)		Function Value
		Name[5]			Type	Range	Type and Range
	Note 3	HFIX	y = (sign of x) • n where n is the largest integer $\leq \lvert x \rvert$	1	REAL*4	any REAL argument	INTEGER*2
Conversion to real	REAL	REAL Note 4		1	INTEGER*4	any INTEGER argument	REAL*4
		Note 1		1	REAL*4	any REAL argument	REAL*4
		SNGL		1	REAL*8	any REAL argument	REAL*4
		SNGLQ		1	REAL*16	any REAL argument	REAL*4
		Note 1	for $z = x_1 + x_2i$, y = REAL(x_1)	1	COMPLEX*8	any COMPLEX argument	REAL*4
		DREAL		1	COMPLEX*16	any COMPLEX argument	REAL*8
		QREAL		1	COMPLEX*32	any COMPLEX argument	REAL*16
Conversion to double	DBLE	DFLOAT		1	INTEGER*4	any INTEGER argument	REAL*8
		DBLE		1	REAL*4	any REAL argument	REAL*8
		Note 1		1	REAL*8	any REAL argument	REAL*8
		DBLEQ		1	REAL*16	any REAL argument	REAL*8
		Note 1	for $z = x_1 + x_2i$, y = DBLE(x_1)	1	COMPLEX*8	any COMPLEX argument	REAL*8
Conversion to extended precision	QEXT	QFLOAT		1	INTEGER*4	any INTEGER argument	REAL*16
		QEXT		1	REAL*4	any REAL argument	REAL*16
		QEXTD		1	REAL*8	any REAL argument	REAL*16
Conversion to complex	CMPLX	Note 1	$y = x_1 + x_2i$ where x_1 = REAL(arg) and $x_2 = 0.$	1	INTEGER*4	any INTEGER argument	COMPLEX*8
		CMPLX		1	REAL*4	any REAL argument	COMPLEX*8
		Note 1		1	REAL*8	any REAL argument	COMPLEX*8
		QCMPLX	$y = x_1 + x_2i$ where x_1 = arg and x_2 = 0.Q0	1	REAL*16	any REAL argument	COMPLEX*32
		Note 1	$y = x_1 + x_2i$ for arg $= x_1 + x_2i$	1	COMPLEX*8	any COMPLEX argument	COMPLEX*8
	Note 3	DCMPLX	$y = x_1 + x_2i$ where x_1 = arg and $x_2 = 0.$	1	REAL*8	any REAL argument	COMPLEX*16

Figure 30 (Part 2 of 3). Conversion and Maximum/Minimum Functions

General Function	Generic Name	Entry Name Name[5]	Definition		Argument(s)		Function Value Type and Range
				Number	Type	Range	
	CMPLX	Note 1	$y = x_1 + x_2i$ where x_1 = REAL(arg1) and x_2 = REAL(arg2)	2	INTEGER*4	any INTEGER argument	COMPLEX*8
		CMPLX		2	REAL*4	any REAL argument	COMPLEX*8
		Note 1		2	REAL*8	any REAL argument	COMPLEX*8
		QCMPLX	$y = x_1 + x_2i$ where x_1 = arg1 and x_2 = arg2	2	REAL*16	any REAL argument	COMPLEX*32
	Note 3	DCMPLX	$y = x_1 + x_2i$ where x_1 = arg1 and x_2 = arg2	2	REAL*8	any REAL argument	COMPLEX*16
Maximum value	MAX	MAX0	$y = \max(x_1,...x_n)$	≥ 2	INTEGER*4	any INTEGER arguments	INTEGER*4
		AMAX1		≥ 2	REAL*4	any REAL arguments	REAL*4
		DMAX1		≥ 2	REAL*8	any REAL arguments	REAL*8
		QMAX1		≥ 2	REAL*16	any REAL argument	REAL*16
	Note 3	AMAX0		≥ 2	INTEGER*4	any INTEGER arguments	REAL*4
	Note 3	MAX1		≥ 2	REAL*4	any REAL arguments	INTEGER*4
Minimum value	MIN	MIN0	$y = \min(x_1,...x_n)$	≥ 2	INTEGER*4	any INTEGER arguments	INTEGER*4
		AMIN1		≥ 2	REAL*4	any REAL argument	REAL*4
		DMIN1		≥ 2	REAL*8	any REAL arguments	REAL*8
		QMIN1		≥ 2	REAL*16	any REAL arguments	REAL*16
	Note 3	AMIN0		≥ 2	INTEGER*4	any INTEGER arguments	REAL*4
	Note 3	MIN1		≥ 2	REAL*4	any REAL arguments	INTEGER*4

Figure 30 (Part 3 of 3). Conversion and Maximum/Minimum Functions

Notes to Figure 30:

[1] No specific name exists for this case. The generic name must be used for this argument type.

[2] IFIX is an alternative specific name for this function.

[3] Specific name must be used to obtain function value of this type.

[4] FLOAT is an alternative specific name for this function.

[5] All functions in all parts of this figure are in-line functions. None of the function names can be passed as arguments. There are no library error codes because there are no library routines.

General Function[2]	Entry Name	Definition	Argument Number	Argument Type	Function Value Type and Range	Error Code
Convert character to integer	ICHAR	Position of character in EBCDIC collating sequence	1	CHARACTER	INTEGER*4	None
Convert integer to character	CHAR	Character corresponding to position of argument in EBCDIC collating sequence	1	INTEGER*4	CHARACTER*1	188
Length of character item	LEN	Length of character entity	1	CHARACTER	INTEGER*4	None
Index of character item	INDEX	Location of substring a_2 in string a_1	2	CHARACTER	INTEGER*4	189, 190
Alphamerically greater than or equal	LGE	$a_1 \geq a_2$ Note 1	2	CHARACTER	LOGICAL*4	191, 192
Alphamerically greater than	LGT	$a_1 > a_2$ Note 1	2	CHARACTER	LOGICAL*4	191, 192
Alphamerically less than or equal	LLE	$a_1 \leq a_2$ Note 1	2	CHARACTER	LOGICAL*4	191, 192
Alphamerically less than	LLT	$a_1 < a_2$ Note 1	2	CHARACTER	LOGICAL*4	191, 192

Figure 31. Character Manipulation Functions

Notes to Figure 31:

[1] Comparison is made using the ASCII collating sequence.

[2] All functions are generated as out-of-line library calls.

General Function	Entry Name	Definition	Number	Argument(s) Type	Range	Function Value Type and Range	Error Code
Logical AND	IAND	k = and (i,j)	2	INTEGER*4	any INTEGER arguments	INTEGER*4	None
Logical OR	IOR	k = or (i,j)	2	INTEGER*4	any INTEGER arguments	INTEGER*4	None
Logical exclusive OR	IEOR	k = xor (i,j)	2	INTEGER*4	any INTEGER arguments	INTEGER*4	None
Logical complement	NOT	k = not (i)	1	INTEGER*4	any INTEGER argument	INTEGER*4	None
Shift operation	ISHFT	k = shift (i,m) i is shifted by m where for m < 0, shift is right; m > 0, shift is left; and m = 0, no shift	2	INTEGER*4	i is any INTEGER argument; $-32 \leq m \leq 32$	INTEGER*4	159
Bit testing and setting	BTEST	1 = bitset (i,m) tests m-th bit of argument i	2	INTEGER*4	i is any INTEGER argument; $0 \leq m \leq 31$ Note 3	LOGICAL*4	159
	IBSET	k = bitset (i,m) sets m-th bit of argument i to 1	2	INTEGER*4		INTEGER*4	159
	IBCLR	k = bitclear (i,m) sets m-th bit of argument i to 0.	2	INTEGER*4		INTEGER*4	159

Figure 32. Bit Manipulation Functions

Notes to Figure 32:

[1] There are no generic names for the bit manipulation functions. All specific names may be passed as actual arguments.

[2] The first four functions are always in-line. The second four are in-line if the second argument is an integer constant; a library function is called if the second argument is an integer variable or expression.

[3] The bits in the first argument (i) are numbered from right to left, beginning at zero. Thus m = 0 corresponds to the right-most bit of the argument i.

Examples of Bit Manipulation Functions

Examples	Explanations
I = IAND (K+3,-1)	The variable K and the constant 3 are added; a logical AND is performed on their sum and the constant -1; the result is stored in variable I.
J = IOR(J,K)	A logical OR is performed of variables J and K; the result is stored in variable J (replacing its previous value).
L = IEOR(5,-1)	A logical exclusive OR is performed on constants 5 and -1; the result is stored in variable L.
M = NOT(J)	A logical NOT is performed on variable J; the result is stored in variable M.

K2 = -5
I = ISHFT(J+K,K2)
	The variables J and K are added; a right shift of 5 bits is performed on their sum; the result is stored in variable I. (Five existing bits were shifted off at the low-order (right) end, and 5 zero bits were shifted in at the high-order (left) end.)

DIMENSION J(5),IAR(10) J(KL)=IBSET (IBCLR (IAR(10),4) ,5)	The value of the tenth element of array IAR is obtained; bit 4 of this value is set to 0; then bit 5 of this value is set to 1; this value is then stored in the element of array J that is identified by the value of variable KL.

Generic		Arguments[1], [4]			Arguments[1], [4]				Function Value	Function Value
Name	Definition	1*4	R*4	R*8	R*16	C*8	C*16	C*32	Type[2]	Length[2]
ABS	Absolute value	x	x	x	x				Argument	Argument
						x	x	x	Real	1/2 Argument
ACOS	Arc cosine		x	x	x				Real	Argument
AINT	Truncation		x	x	x				Real	Argument
ANINT	Nearest whole number		x	x					Real	Argument
ASIN	Arc sine		x	x	x				Real	Argument
ATAN	Arc tangent		x	x	x				Real	Argument
ATAN2	Arc tangent(2 arguments)		x	x	x				Real	Argument
CMPLX	Conversion to complex	x	x	x		x			Complex	8
	Note 3				x				Complex	32
		x	x	x					Complex	8
					x				Complex	32
CONJG	Conjugate					x	x	x	Complex	Argument
COS	Cosine		x	x	x	x	x	x	Argument	Argument
COSH	Hyperbolic cosine		x	x	x				Real	Argument
COTAN	Cotangent		x	x	x				Real	Argument
DBLE	Express as R*8	x	x	x	x	x			Real	8
DIM	Positive difference	x	x	x	x				Argument	Argument
ERF	Error function		x	x	x				Real	Argument
ERFC	1 - Error function		x	x	x				Real	Argument
EXP	Exponentiation		x	x	x	x	x	x	Argument	Argument
GAMMA	Gamma function		x	x					Real	Argument
IMAG	Imaginary part					x	x	x	Real	1/2 Argument
INT	Express as I*4	x	x	x	x	x			Integer	4
LGAMMA	Log of gamma function		x	x					Real	Argument
LOG	Natural logarithm		x	x	x	x	x	x	Argument	Argument
LOG10	Common logarithm		x	x	x				Real	Argument
MAX	Maximum value	x	x	x	x				Argument	Argument
MIN	Minimum value	x	x	x	x				Argument	Argument
MOD	Remainder	x	x	x	x				Argument	Argument
NINT	Nearest integer		x	x					Integer	4
QEXT	Express as R*16	x	x	x					Real	16
REAL	Conversion to real	x	x	x	x				Real	4
						x	x	x	Real	1/2 Argument
SIGN	Transfer of sign	x	x	x	x				Argument	Argument
SIN	Sine		x	x	x	x	x	x	Argument	Argument
SINH	Hyperbolic sine		x	x	x				Real	Argument
SQRT	Square root		x	x	x	x	x	x	Argument	Argument
TAN	Tangent		x	x	x				Real	Argument
TANH	Hyperbolic tangent		x	x	x				Real	Argument

Figure 33. Generic Names for Intrinsic Functions

Notes to Figure 33:

[1] "X" indicates a permissible mode of argument.

[2] "Argument" indicates that the type or length of the result is the same as that of the argument(s).

[3] The specific name DCMPLX must be used to convert an R*8 argument to a C*16 value (or to convert and express two R*8 arguments as a C*16 value).

[4] If more than one argument is permitted, all arguments must be of same type and length.

Part 2. Library Reference

The following topics are discussed in Part 2:

Mathematical, Character, and Bit Routines

Service Subroutines

Data-in-Virtual Subroutines

Extended Error Handling Subroutines and Error Option Table

Multitasking Facility (MTF) Subroutines

For related error messages, see Appendix D, "Library Procedures and Messages" on page 375.

Chapter 6. Mathematical, Character, and Bit Routines

The mathematical, character, and bit routines are supplied by the VS FORTRAN Version 2 library. They perform commonly used computations and conversions. These routines are called either explicitly or implicitly.

Explicitly Called Routines

All the explicitly called routines are intrinsic functions. Each of these functions performs a mathematical, character, or bit manipulation. For detailed information about these functions, see Chapter 5, "Intrinsic Functions" on page 243.

Implicitly Called Routines

The implicitly called routines are used to implement certain FORTRAN operations. The compiler generates the instructions necessary to call the appropriate routine. For example, for the following source statement:

```
ANS = BASE**EXPON
```

where BASE and EXPON are REAL*4 variables, the compiler generates a reference to FRXPR#, the entry name for a routine that raises a real number to a real power.

The implicitly called mathematical and character routines in the VS FORTRAN Version 2 library are described in Figure 35 on page 265 and Figure 36 on page 266. The column headed "Implicit Function Reference" shows a representation of a source statement that might appear in a FORTRAN source module and cause the routine to be called. The rest of the column headings have the same meaning as those used with the explicitly called routines. Implicitly called service routines are in Figure 37 on page 267.

For routines that involve exponentiation, the action taken within a routine depends upon the types of the base and exponent used. Figure 38 on page 267 through Figure 41 on page 268 show the result of an exponentiation performed with the different combinations and values of base and exponent. In these figures, *i* and *j* are integers; *a* and *b* are real numbers; and *c* is a complex number.

Detailed information for calling the routines from assembler language is given in Appendix B, "Assembler Language Information" on page 343.

Alternative Mathematical Library Subroutines

For a small subset of the standard mathematical routines, alternative routines are available. These routines are identical to those in VS FORTRAN Version 1. In VS FORTRAN Version 2, they are referred to as the *alternative mathematical library*. Alternative routines are available for the intrinsic functions ALOG, ALOG10, DLOG, DLOG10, SQRT, DSQRT, CABS, CDABS, SIN, COS, DSIN, DCOS, ACOS, DACOS, ASIN, DASIN, ATAN, ATAN2, DATAN, DATAN2, TAN, COTAN, DTAN, DCOTAN, EXP, and DEXP, and for the implicitly called functions

FDXPD# and FRXPR#. These alternative routines are documented in *VS FORTRAN: Language and Library Reference*, SC26-4119. *VS FORTRAN Version 2 Installation and Customization for VM* and *VS FORTRAN Version 2 Installation and Customization for MVS* describe how these routines can be installed for your use.

Figure 34 shows which libraries contain the various scalar mathematical routines for each version.

Routines	**Version 1 Library**	**Version 2 Library**
New scalar math routines	—	VSF2FORT
Old standard scalar math routines	VFORTLIB	VSF2MATH
Old alternative math routines	VALTLIB	Not available

Figure 34. Libraries Containing Mathematical Routines

General Function	Entry Name[1]	Implicit Function Reference[2]	Argument(s) Number	Argument(s) Type[3]	Function Value Type[3]	Error Code
Multiply and divide complex numbers	CDMPY#	$y = z_1 * z_2$	2	COMPLEX*16	COMPLEX*16	
	CDDVD#	$y = z_1/z_2$	2	COMPLEX*16	COMPLEX * 16	
	CMPY#	$y = z_1 * z_2$	2	COMPLEX*8	COMPLEX*8	
	CDVD#	$y = z_1/z_2$	2	COMPLEX*8	COMPLEX*8	
	CQMPY#	$y = z_1 * z_2$	2	COMPLEX*32	COMPLEX*32	
	CQDVD#	$y = z_1/z_2$	2	COMPLEX*32	COMPLEX*32	
Compare of complex numbers	CXMPR# Note 4	$y = z_1$ compop z_2 Note 5	2	COMPLEX (of all lengths)	LOGICAL*4	
Raise an integer to an integer power	FIXPI#	$y = i ** j$	2	i = INTEGER*4 j = INTEGER*4	INTEGER*4	241
Raise a real number to an integer power	FRXPI#	$y = a ** j$	2	a = REAL*4 j = INTEGER*4	REAL*4	242
	FDXPI#	$y = a ** j$	2	a = REAL*8 j = INTEGER*4	REAL*8	243
	FQXPI#	$y = a ** j$	2	a = REAL*16 j = INTEGER*4	REAL*16	248
Raise a real number to a real power	FRXPR# Note 6	$y = a ** b$	2	a = REAL*4 b = REAL*4	REAL*4	118, 244
	FDXPD# Note 6	$y = a ** b$	2	a = REAL*8 b = REAL*8	REAL*8	119, 245
	FQXPQ#	$y = a ** b$	2	a = REAL*16 b = REAL*16	REAL*16	249 250
Raise 2 to a real power	FQXP2#	$y = 2 ** b$	1	b = REAL*16	REAL*16	260
Raise a complex number to an integer power	FCDXI#	$y = z ** j$	2	z = COMPLEX*16 j = INTEGER*4	COMPLEX*16	247
	FCXPI#	$y = z ** j$	2	z = COMPLEX*8 j = INTEGER*4	COMPLEX*8	246
	FCQXI#	$y = z ** j$	2	z = COMPLEX*32 j = INTEGER*4	COMPLEX*32	270

Figure 35. Implicitly Called Mathematical Routines

Notes to Figure 35:

[1] This name must be used in an assembler language program to call the routine; the # character is a part of the name and must be included.

[2] This is only a *representation* of a FORTRAN statement; it is not the only way the routine may be called.

[3] REAL*4, REAL*8, and REAL*16 arguments correspond to real, double precision, and extended precision arguments, respectively, in VS FORTRAN Version 2.

[4] CXMPR# is an entry name in the library module AFBCCMPR, which is also used for a compare of character arguments.

[5] *compop* is one of the following relational operators: equal or not equal (.EQ. or .NE.).

[6] Available also in the alternative mathematical library.

Entry Name	Implicit Function Reference	Argument(s) Number	Argument(s) Type	Function Value Type	Error Code
CCMPR# Note 2	$y = x_1$ compop x_2 Note 1	6	CHARACTER	Any CHARACTER argument	193 194
CMOVE # Note 2	$y = x$	4	CHARACTER	Any CHARACTER argument	195 196 197
CNCAT#	$y = x_1//x_2. . .x_n//x_n$	≥ 2	CHARACTER	Any CHARACTER argument	198 199

Figure 36. Implicitly Called Character Routines

Notes to Figure 36:

[1] Where *compop* is one of the following relational operators:

equal	.EQ.
not equal	.NE.
greater than	.GT.
less than	.LT.
greater than or equal	.GE.
less than or equal	.LE.

Each character argument implies a pointer to the location and a pointer to the length. The argument list for CCMPR# also has a pointer to the relational operator (*compop*) and a pointer for return of result.

[2] For programs produced by Release 4.0 of the VS FORTRAN Version 1 Compiler, the library functions used for the comparison of character type items and for the assignment of character type items are not invoked. All these operations are performed in-line. These routines remain in the VS FORTRAN Version 2 library to support programs compiled with releases of the compiler earlier than VS FORTRAN Version 1 Release 4.0.

Entry Name	Function	Arguments	Error Code
DSPAN# DSPN2# DSPN4#	Calculate dimension factors and span of adjustable dimension array.	Array description	187
DYCMN#	Obtain storage and relocate adcons for DYNAMIC COMMON.	COMMON and adcon information	156 158

Figure 37. Implicitly Called Service Routines

Base (I)	Exponent (J) J > 0	Exponent (J) J = 0	Exponent (J) J < 0
I > 1	Compute the function value	Function value = 1	Function value = 0
I = 1	Compute the function value	Function value = 1	Function value = 1
I = 0	Function value = 0	Error message 241	Error message 241
I = -1	Compute the function value	Function value = 1	If J is an odd number, function value = -1. If J is an even number, function value = 1.
I < -1	Compute the Function value	Function value = 1	Function value = 0

Figure 38. Exponentiation with Integer Base and Exponent

Base (A)	Exponent (J) J > 0	Exponent (J) J = 0	Exponent (J) J < 0
A > 0	Compute the function value	Function value = 1	Compute the function value
A = 0	Function value = 0	Error message 242 or 243	Error message 242 or 243
A < 0	Compute the function value	Function value = 1	Compute the function value

Figure 39. Exponentiation with Real Base and Integer Exponent

Base (A)	Exponent (B) B > 0	Exponent (B) B = 0	Exponent (B) B < 0
A > 0	Compute the function value	Function value = 1	Compute the function value
A = 0	Function value = 0	Error message 244 or 245	Error message 244 or 245
A < 0	Error message 118 or 119	Function value = 1	Error message 118 or 119

Figure 40. Exponentiation with Real Base and Exponent

Base (C) $C = P + Qi$	**Exponent (J)** $J > 0$	**Exponent (J)** $J = 0$	**Exponent (J)** $J < 0$
$P > 0$ and $Q > 0$	Compute the function value	Function value $= 1 + 0\,i$	Compute the function value
$P > 0$ and $Q = 0$	Compute the function value	Function value $= 1 + 0\,i$	Compute the function value
$P > 0$ and $Q < 0$	Compute the function value	Function value $= 1 + 0\,i$	Compute the function value
$P = 0$ and $Q > 0$	Compute the function value	Function value $= 1 + 0\,i$	Compute the function value
$P = 0$ and $Q = 0$	Function value $0 + 0\,i$	Error message 246 or 247	Error message 246 or 247
$P = 0$ and $Q < 0$	Compute the function value	Function value $= 1 + 0\,i$	Compute the function value
$P < 0$ and $Q > 0$	Compute the function value	Function value $= 1 + 0\,i$	Compute the function value
$P < 0$ and $Q = 0$	Compute the function value	Function value $= 1 + 0\,i$	Compute the function value
$P < 0$ and $Q < 0$	Compute the function value	Function value $= 1 + 0\,i$	Compute the function value

Figure 41. Exponentiation with Complex Base and Integer Exponent

Chapter 7. Service Subroutines

The VS FORTRAN Version 2 Library provides subroutines for general programming tasks. The subroutines are called by the appropriate entry name in a CALL statement. The following VS FORTRAN Version 2 service routines are described in this chapter:

Mathematical Exception Test Subroutines

DVCHK	Tests for divide-check exception
OVERFL	Tests for exponent overflow or underflow

Storage Dump Subroutines

DUMP/PDUMP	Provides a symbolic dump of a specified area of storage
CDUMP/CPDUMP	Provides a symbolic dump of a specified area of storage containing character data
SDUMP	Provides a symbolic dump of all variables in a program unit

Return Code Subroutines

SYSRCS	Saves a return code value for future termination
SYSRCT	Obtains the value of the currently saved return code
SYSRCX	Ends processing of the program using either the saved return code or a supplied return code

Other Service Subroutines

ASSIGNM	Moves a character string containing double-byte data to a character variable, substring or array element, preserving balanced shift codes
CLOCK/CLOCKX	Provides the value of the time-of-day clock
CPUTIME	Lets you determine the amount of CPU time used by a program or a portion of a program
DATIM/DATIMX	Provides extended information about the date and time
EXIT	Ends processing of the program
FILEINF	Sets up file characteristics that will be used by an OPEN or an INQUIRE statement
SYSABN/SYSABD	Specifies abnormal termination of your job, with or without an accompanying storage dump
UNTNOFD/UNTANY	Identifies FORTRAN unit numbers that are available
XUFLOW	Allows or suppresses a program interrupt caused by exponent underflow

For information on data-in-virtual subroutines, see Chapter 8, "Data-in-Virtual Subroutines" on page 293.

For information on extended error-handling subroutines, see Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.

For information on multitasking facility subroutines, see Chapter 10, "Multitasking Facility (MTF) Subroutines" on page 335.

Mathematical Exception Test Subroutines

These routines test the status of indicators and may return a value to the calling program. In the following description of the routines, *k* represents an integer value.

DVCHK Subroutine

The DVCHK subroutine tests for a divide-check exception and returns a value indicating the existing condition.

> **Syntax**
>
> **CALL DVCHK (*k*)**

k
  is an integer or real variable in the program unit.

The values of *k* returned have the following meanings:

Value	Meaning
1	The divide-check indicator is *on*.
2	The divide-check indicator is *off*.

OVERFL Subroutine

The OVERFL subroutine tests for exponent overflow or underflow, and returns a value indicating the existing condition. After testing, the overflow indicator is turned off.

> **Syntax**
>
> **CALL OVERFL (*k*)**

k
  is an integer variable defined within this program unit.

The values of *k* returned have the following meanings:

Value	Meaning
1	Floating-point overflow occurred last.
2	No overflow or underflow condition is current.
3	Floating-point underflow occurred last.

Note: The values for 1 and 3 indicate the last one to occur; if the same statement causes an overflow followed by an underflow, the value returned is 3 (underflow occurred last).

Storage Dump Subroutines

DUMP/PDUMP Subroutines

The DUMP/PDUMP subroutine dynamically dumps a specified area of storage onto the system output data set. When you use DUMP, the processing stops after the dump; when you use PDUMP, the processing continues after the dump.

Syntax

CALL {DUMP | PDUMP} (a_1,b_1,k_1,a_2,b_2,k_2,...)

a and *b*

are variables in the program unit. Each indicates an area of storage to be dumped.

Either *a* or *b* can represent the upper or lower limit of the storage area.

k

specifies the dump format to be used.

The values that can be specified for *k* and their meanings are:

Value	Format Requested
0	Hexadecimal
1	LOGICAL*1
2	LOGICAL*4
3	INTEGER*2
4	INTEGER*4
5	REAL*4
6	REAL*8
7	COMPLEX*8
8	COMPLEX*16
9	CHARACTER
10	REAL*16
11	COMPLEX*32

Programming Considerations for DUMP/PDUMP

A load module or phase may occupy a different area of storage each time it is executed. To ensure that the appropriate areas of storage are dumped, the following conventions should be observed.

If an array and a variable are to be dumped at the same time, a separate set of arguments should be used for the array and for the variable. The specification of limits for the array should be from the first element in the array to the last element. For example, assume that A is a variable in common, B is a real number, and TABLE is an array of 20 elements. The following call to the storage dump routine could be used to dump TABLE and B in hexadecimal format, and stop the program after the dump is taken:

```
CALL DUMP(TABLE(1),TABLE(20),0,B,B,0)
```

If an area of storage in common is to be dumped at the same time as an area of storage not in common, the arguments for the area in common should be given separately. For example, the following call to the storage dump routine could be used to dump the variables A and B in REAL*8 format without stopping the program:

```
CALL PDUMP(A,A,6,B,B,6)
```

If variables not in common are to be dumped, each variable must be listed separately in the argument list. For example, if R, P, and Q are defined implicitly in the program, the statement

```
CALL PDUMP(R,R,5,P,P,5,Q,Q,5)
```

should be used to dump the three variables in REAL*4 format. If the statement

```
CALL PDUMP(R,Q,5)
```

is used, all main storage between R and Q is dumped, which may or may not include P, and may include other variables.

If an array and a variable are passed to a subroutine as arguments, the arguments in the call to the storage dump routine in the subroutine should specify the parameters used in the definition of the subroutine. For example, if the subroutine SUBI is defined as:

```
SUBROUTINE SUBI(C,Y)
DIMENSION X (10)
```

and the call to SUBI within the source module is:

```
DIMENSION A (10)
      .
      .
      .
CALL SUBI (A,B)
```

then the following statement should be used in SUBI to dump the variables in hexadecimal format without stopping the program:

```
CALL PDUMP (X(1),X(10),0,Y,Y,0)
```

CDUMP/CPDUMP Subroutines

The CDUMP/CPDUMP subroutine dynamically dumps a specified area of storage containing character data. When you use CDUMP, the processing stops after the dump; when you use CPDUMP the processing continues after the dump.

Syntax

CALL {CDUMP | CPDUMP} (a_1,b_1,a_2,b_2,...)

a and *b*
variables in the program unit. Each indicates an area of storage to be dumped.

Either *a* or *b* can represent the upper or lower limit of each storage area.

The dump is always produced in character format. (A dump format type (as for DUMP/PDUMP) must not be specified.)

Programming Considerations for CDUMP/CPDUMP

A load module may occupy a different area of storage each time it is executed. To ensure that the appropriate areas of storage are dumped, the following conventions should be observed.

If an array and a variable are to be dumped at the same time, a separate set of arguments should be used for the array and for the variable. The specification of limits for the array should be from the first element in the array to the last element. For example, assume that A is a variable in common, B is a real number, and TABLE is an array of 20 elements. The following call to the storage dump routine could be used to dump TABLE and B in hexadecimal format, and stop the program after the dump is taken:

```
CALL CDUMP(TABLE(1),TABLE(20),0,B,B,0)
```

SDUMP Subroutine

The SDUMP subroutine provides a symbolic dump that is displayed in a format dictated by variable type as coded or defaulted in your source. Data is dumped on the error message unit. The symbolic dump is created by program request, on a program unit basis, using CALL SDUMP. (In addition, variables can be dumped automatically upon abnormal termination using the compile-time option SDUMP. For more information on the compile-time option, see *VS FORTRAN Version 2 Programming Guide.*)

Items displayed are:

- All referenced, local, named variables in their FORTRAN-defined data representation
- All variables contained in a blank common, named common, or a dynamic common area in their FORTRAN-defined data representation
- Nonzero or nonblank character array elements only
- Array elements with their correct indexes

Note that the amount of output produced can be very large, especially if your program has large arrays, or large arrays in common blocks. For such programs, you may want to avoid calling SDUMP.

Syntax

CALL SDUMP [(*rtn*$_1$,*rtn*$_2$,...)]

rtn$_1$,*rtn*$_2$,...
are names of other program units from which data will be dumped. These names must be listed in an EXTERNAL statement.

Programming Considerations for SDUMP

- To obtain symbolic dump information and location of error information, compilation must be done either with the SDUMP option or with the TEST option.
- Calling SDUMP and specifying program units that have not been entered gives unpredictable results.
- Calling SDUMP with no parameters produces the symbolic dump for the current program unit:

```
CALL SDUMP
```

- An EXTERNAL statement must be used to identify the names being passed to SDUMP as external routine names.
- At higher levels of optimization (1, 2, and 3), the symbolic dump may show incorrect values for some variables because of compiler optimization techniques.
- Values for uninitialized variables are unpredictable. Arguments in uncalled subprograms or in subprograms with argument lists shorter than the maximum may cause the SDUMP subroutine to fail.
- The display of data can also be invoked automatically. If the run-time option ABSDUMP is in effect and your program abends (abnormally termi-

nates) in a program unit compiled with the SDUMP option or with the TEST option, all data in that program unit is automatically dumped.

Additionally, all data in any program unit in the save area traceback chain compiled with the SDUMP option or with the TEST option is also dumped. Data occurring in a common block is dumped at each occurrence, because the data definition in each program unit may be different. The display of data follows the AFB240I message and the traceback messages on the object time error unit.

Examples follow of calling SDUMP from the main program and from a subprogram.

In the main program, the statement

```
EXTERNAL PGM1,PGM2,PGM3
```

would make the address of subprograms PGM1, PGM2, and PGM3 available for a call to SDUMP:

```
CALL SDUMP (PGM1,PGM2, PGM3)
```

that would cause variables in PGM1, PGM2, and PGM3 to be printed.

In PGM1, the statement

```
EXTERNAL PGM2,PGM3
```

makes PGM2 and PGM3 available. (PGM1 is missing because the call is in PGM1.)

The statements

```
CALL SDUMP
CALL SDUMP (PGM2,PGM3)
```

will dump variables for PGM1, PGM2, and PGM3.

For information about output from symbolic dumps, see *VS FORTRAN Version 2 Programming Guide.*

Return Code Subroutines

The return code subroutines let you manipulate a return code that will be issued when your application program terminates normally. The CALL EXIT or CALL SYSRCX statement stops the program and supplies the saved value of the return code to the operating system.

SYSRCS Subroutine

You can set or modify the saved value of the return code by issuing a call to SYSRCS. The initial value of the saved return code is 0.

Syntax

CALL SYSRCS (*n*)

n

is an integer expression that must be within the range 0 to 4095, inclusive. This value is saved and used as the return code for completion of the program when a CALL EXIT is issued. This value will also be returned by SYSRCX if no parameter is specified there.

SYSRCT Subroutine

SYSRCT obtains the value of the currently saved return code. The initial value of the saved return code is 0.

Syntax

CALL SYSRCT (*m*)

m

is an integer variable. SYSRCT stores the current value of the saved return code into the integer variable m, and returns to the calling program.

SYSRCX Subroutine

SYSRCX stops the processing using either the saved return code or the return code supplied in this call (*k*). The initial value of the supplied return code is 0.

Syntax

CALL SYSRCX [(*k*)]

k

is an integer expression that must be within the range 0 to 4095, inclusive. If *k* is specified, processing stops normally, and control is returned to the operating system with the return code specified by *k*. Any previously saved return code is ignored.

If no parameter is specified, the processing is stopped normally. If a return code was set by a previous call to SYSRCS, control is returned to the operating system with the saved return code. If no return code was previously set, control is returned to the operating system with a return code of 0.

Other Service Subroutines

ASSIGNM Subroutine

The ASSIGNM subroutine will move a character string containing double-byte data to a character variable, substring or array element, preserving the balanced shift codes.

Syntax

CALL ASSIGNM (*input*, *output*, *rcode*, *rsncode*)

input
is a character variable, array element, or character expression containing the characters to be moved.

output
is a character variable, character substring or character array element in which is placed the moved characters.

rcode
is an integer variable or array element of length 4 which will contain the return code from the ASSIGNM subroutine.

rsncode
is an integer variable or array element of length 4 which will contain the reason code from the ASSIGNM subroutine.

Return Codes	Reason Codes	Explanation
0	0	Successful completion
4	3	Warning; output string truncated
8	120	Error; Output string overlaps input string Error is detected before data is moved; output is never performed.
12	No value returned	Severe Error, required parameter is missing

Programming Considerations for ASSIGNM

The input string will be moved to the output storage location. The length of the input string and the output area need not be the same. If the input string is shorter than the output area, the input string will be moved to the output area and padded on the right with EBCDIC blank characters (X'40'). If the input string is longer than the output area, the input string will be truncated and moved to the output area. Truncation will occur using the following rules:

- EBCDIC characters will be truncated at any position
- Double-byte characters will be truncated after the second byte of the double-byte character. A shift-in character will be added after the double-byte character.

If the truncation will not leave enough room for a shift-in character to be added, the last double-byte character will be truncated and a shift-in character plus a pad character will be added to the output string.

Figure 42 shows how ASSIGNM pads the output string and Figure 43 shows how ASSIGNM truncates the output string.

```
Column: 1    6
        ---------------------------------------------------------------------
             CHARACTER*10 DBCS_STUFF
             CHARACTER*15 STUFF
             INTEGER*4    RTCODE,RSNCODE
             DBCS_STUFF = 'HOUSE<kk>'
             CALL ASSIGNH (DBCS_STUFF,STUFF,RTCODE,RSNCODE)
        *  These are the results of the CALL:
        *  STUFF = HOUSE<kk>bbbbbb    (where "b" is a blank character)
        *  RTCODE = 0
        *  RSNCODE = 0                (indicates padding with EBCDIC blanks)
```

Figure 42. Example of ASSIGNM Padding

```
Column: 1    6
        ---------------------------------------------------------------------
             CHARACTER*24 DBCS_STUFF
             CHARACTER*14 STUFF1
             CHARACTER*11 STUFF2
             INTEGER*4    RTCODE,RSNCODE
             DBCS_STUFF = '<.W.H.A.T  .A  .L.I.F.E>'
             CALL ASSIGNH (DBCS_STUFF,STUFF1,RTCODE,RSNCODE)
        *  These are the results of the CALL:
        *  STUFF1 = <.W.H.A.T  .A>
        *  RTCODE = 4
        *  RSNCODE = 3                (indicates truncating)
        *
             CALL ASSIGNH (DBCS_STUFF,STUFF2,RTCODE,RSNCODE)
        *  These are the results of the CALL:
        *  STUFF2 = <.W.H.A.T>b       (where "b" is a blank character)
        *  RTCODE = 4
        *  RSNCODE = 3                (indicates truncating)
```

Figure 43. Example of ASSIGNM Truncation

CLOCK/CLOCKX Subroutines

The values of CLOCK and CLOCKX are derived from the time-of-day clock. The values are returned in INTEGER*4 variables (to the nearest second) for CLOCK, or in REAL*8 variables for CLOCKX.

Note: You cannot use the value returned in CLOCK or CLOCKX to derive the date or time returned from DATIM or DATIMX. The value of the abbreviated time-of-day clock is not synchronized with the date and time values.

CLOCK

CLOCK returns values of the time-of-day clock to the nearest second. The least significant bit of the time-of-day clock value returned by CLOCK is incremented every 1.048576 seconds.

CLOCK conforms to Industrial Real Time FORTRAN (IRTF) standards.

Syntax

CALL CLOCK (*cpuclk* **[,***count* **[,***max* **]])**

cpuclk
is the value of the time-of-day clock expressed as a positive integer.

count
is the amount by which the time-of-day clock value is increased per second. The clock value is incremented by 1 for every second, so the count is 1.

max
is the maximum value of the INTEGER*4 time-of-day clock. (In hexadecimal, this value is 7FFFFFFF.)

CLOCKX

CLOCKX returns an abbreviated version of the time-of-day clock in a REAL*8 variable. The least significant bit of this abbreviated version is incremented every microsecond.

Syntax

CALL CLOCKX (*cpuckx* **[,***xcount* **[,***xmax* **]])**

cpuckx
is the value of the time-of-day clock expressed as a REAL*8 variable.

xcount
is the amount by which the time-of-day clock value is increased per second. The clock value is incremented by 1000000 for every second, so *xcount* is always 1000000.

xmax
is the maximum value of the REAL*8 time-of-day clock. (In hexadecimal, this value is 4D7FFFFFFFFFFFF0.)

CPUTIME Subroutine

The CPUTIME subroutine allows you to determine how much CPU time a program or a portion of a program has used.

Syntax

CALL CPUTIME (*accumcpu*, *rcode*)

accumcpu
: a real variable or array element of length 8 in which is placed a value representing the number of microseconds of CPU time that has accumulated since some arbitrary base. This base generally remains unchanged across successive CPUTIME calls (a non-zero return code indicates otherwise).

On VM, the CPU time returned by the subroutine is in virtual processor time; that is, the time used directly by the user's virtual machine. On MVS, the CPU time returned by the subroutine is in task time; that is, the time accumulated while the task is in execution

rcode
: an integer variable or array element of length 4 in which is placed the return code upon return from CPUTIME.

Return Code	Explanation
0	Successful completion. The value in *accumcpu* can be used for either of the following: ► As a starting value ► As a value for computing the CPU time used since a previous CPUTIME call
4	Accumulated value reset. The value in *accumcpu* can be used only as a starting value. This situation occurs, on MVS only, when the timing information base is changed by an Interactive Debug function that is no longer in use.
8	No accumulated value. *accumcpu* becomes undefined and its value should not be used. This situation occurs, on MVS only, when the timing information base is changed by an Interactive Debug function that is now in use.

Programming Considerations for CPUTIME:

To obtain the amount of CPU time used in a portion of a program, simply follow these steps:

1. Code two CPUTIME calls, one before the portion of the program and one after it.
2. Calculate the difference between the values returned by the two calls.

An example of calculating the CPU time used by a portion of a program follows.

Using CPUTIME Calls:

```
Column:  1    6
         ----------------------------------------------------------------
               REAL*8  ACCUM_A, ACCUM_B, USED_TIME
               .
               .
               .
               CALL CPUTIME (ACCUM_A, IRCODE_A)            ! Call "A"
               .
               .  (Portion of the program you are interested in)
               .
               CALL CPUTIME (ACCUM_B, IRCODE_B)            ! Call "B"
         *
         *  Calculating the CPU time used
         *
               IF (IRCODE_A .NE. 8 .AND. IRCODE_B .EQ. 0) THEN
                   USED_TIME = ACCUM_A - ACCUM_B
                   PRINT *, USED_TIME,' microseconds of CPU time were used.'
               END IF
               .
               .
               .
               END
```

MVS Considerations:

When using Interactive Debug under MVS, the CPUTIME subroutine will **not** provide information during any of the following situations:

- While a program unit is being timed (TIMER command)
- While program sampling is in effect (SAMPLE option of ENDDEBUG command)
- While animation is in progress (STEP command)

If CPUTIME is invoked and one of these Interactive Debug functions either is being used or has been used, a non-zero return code indicates that the accumulation of CPU timing information has been interrupted and the base has been changed.

DATIM/DATIMX Subroutines

The date and time routine provides the current local date and time. To obtain date and time information, you need to call either DATIM or DATIMX. The time value is accurate to the nearest hundredth of a second for MVS, and to the nearest second for VM.

DATIM

DATIM provides information about the date, time of day, and processor clock. DATIM conforms to Industrial Real Time FORTRAN (IRTF) standards.

The processor clock value is provided in an abbreviated version, to the nearest second. The least significant bit of this abbreviated version is incremented every 1.048576 seconds.

Syntax

CALL DATIM(*now*)

now

is an integer array of at least 8 INTEGER*4 values. The values returned in the first 8 elements of the array are as follows:

Element	Contents
1	The value of the clock expressed as a positive integer. A value of -1 indicates that the clock is invalid.
2	Milliseconds (0-990) precise to the hundredths position. (For MVS only. Under CMS, the value of time is accurate to the nearest second.)
3	Seconds (0-59)
4	Minutes (0-59)
5	Hour using a 24-hour clock (0-23)
6	Day of the month (1-31)
7	Month of the year (1-12)
8	Year (4 digits, for example: 1986)

DATIMX

DATIMX provides you with date and time information that can be used by your program to produce printable or formatted data.

DATIMX obtains values for the processor clock, milliseconds, second, minute, hour on a 24-hour clock, day of the month, month of the year, 4-digit year, hour on a 12-hour clock with AM or PM indicator, day of the week, day of the year, and 2-digit year. You can modify the presentation style of the date and time information to suit your needs.

The processor clock value is provided in an abbreviated version. The least significant bit of this abbreviated version is incremented every 1.048576 seconds.

Syntax

CALL DATIMX(*now*)

now

is an integer array of at least 14 INTEGER*4 values. The values returned in the first 14 arguments of the array are as follows:

Element	Contents
1	The value of the clock expressed as a positive integer. A value of -1 indicates that the clock is invalid.
2	Milliseconds (0-990) precise to the hundredths position. (For MVS only. Under CMS, the value of time is accurate to the nearest second.)
3	Seconds (0-59)
4	Minutes (0-59)
5	Hour using a 24-hour clock (0-23)
6	Day of the month (1-31)
7	Month of the year (1-12)
8	Year (4 digits, for example: 1986)
9	Reserved. Value returned is -1.
10	Hour using a 12-hour clock (1-12)
11	AM/PM indicator. 1 is returned for AM, 2 for PM.
12	Day of the week (1-7, beginning with Sunday)
13	Day of the year (1-366)
14	Year (2 digits, for example: 86)

EXIT Subroutine

The EXIT subroutine terminates the executable program and returns control to the operating system.

Syntax

CALL EXIT

CALL EXIT performs a function similar to that of the STOP statement, except that no operator message is produced.

FILEINF Subroutine

The FILEINF subroutine can be used to set up the file characteristics prior to issuing an OPEN or an INQUIRE statement.

Syntax

CALL FILEINF [(*rcode* **[,** *param1*, *value1*, *param2*, *value2*, ...**])]**

rcode

a four-byte integer variable or array element in which is placed the return code upon return from FILEINF. If coded, *rcode* must appear first in the parameter list.

Return Code	Explanation
0	Successful completion
4	Argument list is in incorrect format
8	Argument list contains invalid keyword parameter
12	Parameter DEVICE has an incorrect value
16	Parameter VOLCNT has an incorrect value
20	Parameter VOLSER has an incorrect value
24	Parameter VOLSERS has an incorrect value
28	Parameter CYL has an incorrect value
32	Parameter TRK has an incorrect value
36	Parameter MAXBLK has an incorrect value
40	Parameter MAXREC has an incorrect value
44	Parameter SECOND has an incorrect value
48	Parameter DIR has an incorrect value
52	Parameter RECFM has an incorrect value
56	Parameter LRECL has an incorrect value
60	Parameter BLKSIZE has an incorrect value

param
a character expression whose value when any trailing blanks are removed can be one of the following:

DEVICE	CYL | TRK | MAXBLK | MAXREC	RECFM
VOLCNT	SECOND	LRECL
VOLSER	DIR	BLKSIZE
VOLSERS		

value
either a character expression, a character array, or an integer expression, depending on the parameter to which it corresponds.

DEVICE The *value* that corresponds to DEVICE is a character expression (1 to 8 characters long) whose value, when any trailing blanks are removed, is the type of the device. It can be the unit address such as 123, an IBM-supplied name such as 3380, or a user-assigned group name such as SYSDA.

VOLCNT The *value* that corresponds to VOLCNT is an integer expression of length 4. The expression's value specifies a maximum number of volumes an output data set requires (valid number is from 1 to 255).

VOLSER The *value* that corresponds to VOLSER is a character expression whose value, when trailing blanks are removed, is a volume serial number with a length of 1 to 6 characters. Valid characters include letters, numbers, national characters ($, #, @) and the hyphen. Special characters are not allowed.

VOLSERS This parameter is used when there are more than one volume serial numbers needed. The maximum number of serial numbers is 225. The *value* that corresponds to VOLSERS is a character array whose elements, when trailing blanks are removed, are volume serial numbers with lengths of 1 to 6 characters. Each element in the array contains a volume serial number, except for the last element, which must contain an asterisk (*) in the first position to indicate the end of the list.

If you want to specify two different volume serial numbers, for example, *volser1* and *volser2*, the correct way to do it is as follows:

```
DIMENSION ARRAY(3)
ARRAY(1) = 'volser1'
ARRAY(2) = 'volser2'
ARRAY(3) = '*'
CALL FILEINF (..., 'VOLSERS', ARRAY, ...)
```

Duplicate volume serial numbers and special characters are not allowed.

CYL | TRK | MAXBLK | MAXREC

These parameters are mutually exclusive. The *value* that corresponds to them is an integer expression of length 4. The value of the integer expression is the amount of primary space required to allocate the new data set. This space information may be in cylinders, tracks, blocks, or records.

If MAXBLK is specified, the value specified or defaulted for BLKSIZE will become the block length. If MAXREC is specified, the value will be converted into blocks. If this parameter is omitted, the space information will be obtained from the unit attribute table.

SECOND The *value* that corresponds to SECOND is an integer expression of length 4. The expression's value is the amount of additional space which will be allocated if more space is needed to create a new data set.

DIR The *value* that corresponds to DIR is an integer expression of length 4. Its value is the number of 256-byte records to be contained in the directory of a new partitioned data set. If this parameter is omitted, and the FILE specifier on the OPEN statement refers to a member of a new partitioned data set, a value of 5 will be used.

RECFM The *value* that corresponds to RECFM is a character expression whose value, when trailing blanks are removed, must be F, FA, FB, FBA, V, VA, VB, VBA, VS, VBS, U or UA. RECFM specifies the record format of the file connected for sequential access. Direct access files always have a record format of F. If RECFM is not specified, the value from the default attribute table will be used.

LRECL The *value* that corresponds to LRECL is an integer expression of length 4. The expression's value is the logical record length of the file. If the record format (RECFM) is variable, the record length (LRECL) has to include 4 bytes for the record length field. A value of -1 indicates that the logical record length is unlimited.

BLKSIZE The *value* corresponding to BLKSIZE is an integer expression of length 4. The expression's value is the block size length of the file connected for sequential access. The valid block size range is from 1 to 32760. If BLKSIZE is not specified for a file connected for sequential access, the value from the default attribute table will be used. If the record format is not blocked and the block size is specified, BLKSIZE will be ignored.

Programming Considerations for FILEINF

The parameters can be specified in any order and the CALL can contain none, some or all of the parameters. If no parameters are used, the file's space and DCB information will be obtained from the unit attribute table instead. For more information, see *VS FORTRAN Version 2 Programming Guide*.

If any parameter is specified more than once on the CALL, the last one will be used. This rule also applies to the mutually exclusive parameters CYL, TRK, MAXBLK, and MAXREC.

If an integer parameter is given the value of zero or a character parameter is given the value of blanks, it will be treated as if the parameter were not coded on the CALL.

The information provided on the CALL will be applied to a certain OPEN or INQUIRE statement that follows the CALL. The following OPEN and INQUIRE statements use the information from the FILEINF routine:

- Any OPEN or INQUIRE that indicates an MVS data set name or a CMS file identifier on the FILE specifier. (Information provided on the CALL will *not* be used for an OPEN or INQUIRE that specifies a ddname.)
- Any OPEN that specifies STATUS='SCRATCH' when there is no explicit file definition for that file.

The information provided on the FILEINF call is available only for the OPEN or INQUIRE statement that immediately follows it. To specify file information for a subsequent OPEN or INQUIRE statement, you must code another FILEINF call. READ, WRITE, and other FORTRAN statements do not use any of the information given on the CALL.

Multiple FILEINF Calls: If FILEINF is called with just the return code or without any parameters, all of the information set up by the previous call will become ineffective.

If FILEINF is called with any parameters, all of the information set up by the previous call will be replaced.

VM Considerations: Only the following parameters are applicable to VM: MAXREC, RECFM, LRECL, and BLKSIZE. If the remaining parameters are specified, they will be ignored. The values corresponding to those parameters will not be verified, but the keywords will be verified.

MVS Considerations: For INQUIRE statements on MVS, only DEVICE, VOLSER (or VOLSERS), and RECFM are meaningful. If you specify the remaining parameters, they will be verified and ignored.

Error Conditions: If an error is detected in the CALL, the following OPEN or INQUIRE statement that is coded to use the FILEINF information will get an error message also, and the statement will be ignored.

Examples of Valid FILEINF Calls

Example 1:

```
Column:   1    6
          ----------------------------------------------------------------
                CALL FILEINF( IRCODE, 'TRK', 20, 'SECOND', 10, 'DIR', 5,
               1     'RECFM', 'FB', 'LRECL', 80, 'BLKSIZE', 3200 )
```

Example 2:

```
Column:   1    6
          ----------------------------------------------------------------
                CHARACTER*10  DEV, VOL
                DEV='3380'
                VOL='J76VOL'
                CALL FILEINF( IRCODE, 'DEVICE', DEV, 'VOLSER', VOL,
               1     'RECFM', 'FB', 'LRECL', 80, 'BLKSIZE', 3200 )
```

SYSABN/SYSABD Subroutines

The two abend routines allow you to specify abnormal termination of your job, with or without an accompanying storage dump. If the run-time option STAE is in effect, the requested abnormal termination occurs after the message AFB240I, the traceback, and, optionally, the post-abend dump are printed. If NOSTAE is in effect, the job is terminated immediately.

SYSABN

SYSABN causes abnormal termination of your job without a dump.

Syntax

CALL SYSABN (*compl-code*)

compl-code
is an INTEGER*4 expression used as the user completion code when the abend occurs. Valid values are:

```
   1 through  239
 241 through  499
 501 through  899
1000 through 4095
```

(Completion code values 240, 500, and 900 through 999 are reserved for VS FORTRAN Version 2.)

SYSABD

SYSABD causes abnormal termination of your job with a dump. A SYSUDUMP DD statement is required to produce the dump. If the DD statement is not present, the abend occurs without a dump.

Syntax

CALL SYSABD (*compl-code*)

compl-code
is an INTEGER*4 expression used as the user completion code when the abend occurs.

Valid values are:

```
   1 through  239
 241 through  499
 501 through  899
1000 through 4095
```

(Completion code values 240, 500, and 900 through 999 are reserved for VS FORTRAN Version 2.)

UNTNOFD/UNTANY Subroutines

The unit checking subroutines allow you to identify the FORTRAN unit numbers that are available, within a specified range of unit numbers. For these subroutines, an **available** unit number is one that is:

- not currently connected to any file, **or**
- preconnected, but for which no I/O statements other than INQUIRE have been issued.

UNTNOFD returns the lowest available unit number that does not have a file definition; UNTANY returns the lowest available unit number, regardless of the file definitions in effect.

UNTNOFD

Within a specified range, UNTNOFD will return the lowest unit number that does not have a user-specified file definition associated with it (with a ddname of FT*nn*F001 or FT*nn*K01, where *nn* is the unit number) and that is available.

Syntax

CALL UNTNOFD (*rcode*, *startnum*, *endnum*, *unitnum***)**

rcode
a four-byte integer variable or array element that will contain the return code upon return from UNTNOFD.

Return Code	Explanation
0	Successful completion. A unit number is being returned.
8	No unit within the specified range meets the criteria.
12	The argument list has an incorrect number of arguments.
16	The value specified for *startnum* exceeds the largest allowable unit number, or is a negative value.
20	The value specified for *endnum* exceeds the largest allowable unit number.
24	The value specified for *endnum* is smaller than the value specified for *startnum*.

startnum
a four-byte integer expression that specifies the first unit number of a range of unit numbers. The value of *startnum* must be zero or a positive integer less than the largest unit number set up by the installation.

endnum
a four-byte integer expression that specifies the last unit number of a range of unit numbers. The value of *endnum* may be one of the following:

- a positive integer greater than *startnum* but less than or equal to the largest unit number set up by the installation, or
- a negative value, which indicates the largest unit number set up by the installation.

unitnum
a four-byte integer variable or array element in which is placed the value of the lowest unit number that does not have a file definition in effect and that is available, within the range specified by *startnum* and *endnum*. If no unit number within the range meets the criteria, *unitnum* becomes undefined.

The following unit numbers will never be returned by UNTNOFD:

- The standard I/O unit number for the error messages and PRINT/WRITE statements (usually unit 6)
- On VM, the standard I/O unit numbers for the reader and punch (usually units 5 and 7, respectively)

If no parameters are specified, the call will be ignored.

Examples of Valid UNTNOFD Calls

Example 1:

Assume the following:

- There is a file definition in effect with the ddname FT01F001.
- There is no file definition in effect with ddname FT02F001 or FT02K01.
- No I/O statements have been issued.

After the following UNTNOFD call:

```
CALL UNTNOFD (IRCODE, 1, 10, IUNIT)
```

IUNIT will contain the value 2, since unit 2 is the lowest unit within the range (from 1 to 10) that does not have a file definition, that is not currently connected to a file, and for which no I/O statements have been issued.

Example 2:

The following UNTNOFD call will identify the lowest unit, in the range from 50 to the largest unit number allowed, that does not have a file definition in effect and that is available.

```
CALL UNTNOFD (IRCODE, 50, -1, IUNIT)
```

UNTANY

Within a specified range, UNTANY will return the lowest unit number of a unit that is available, regardless of the file definitions in effect.

Syntax

CALL UNTANY (*rcode*, *startnum*, *endnum*, *unitnum***)**

rcode
a four-byte integer variable or array element which will contain the return code upon return from UNTANY.

Return Code	Explanation
0	Successful completion. A unit number is being returned.
8	No unit within the specified range meets the criteria.
12	The argument list has an incorrect number of arguments.
16	The value specified for *startnum* exceeds the largest allowable unit number, or is a negative value.
20	The value specified for *endnum* exceeds the largest allowable unit number.
24	The value specified for *endnum* is smaller than the value specified for *startnum*.

startnum
a four-byte integer expression that specifies the first unit number of a range of unit numbers. The value of *startnum* must be zero or a positive integer less than the largest unit number set up by the installation.

endnum
a four-byte integer expression that specifies the last unit number of a range of unit numbers. The value of *endnum* may be one of the following:

- a positive integer greater than *startnum* but less than or equal to the largest unit number set up by the installation, or
- a negative value, which indicates the largest unit number set up by the installation.

unitnum
a four-byte integer variable or array element in which is placed the value of the lowest unit number that is available, if any, within the range specified by *startnum* and *endnum*. If no unit number within the range meets the criteria, *unitnum* becomes undefined.

The following unit numbers will never be returned by UNTANY:

- The standard I/O unit number for the error messages and PRINT/WRITE statements (usually unit 6)
- On VM, the standard I/O unit numbers for the reader and punch (usually units 5 and 7, respectively)

If no parameters are specified, the call will be ignored.

Examples of Valid UNTANY Calls

Example 1:

Assume the following:

- There is a file definition in effect with the ddname FT01F001.
- There is no file definition in effect with ddname FT02F001 or FT02K01.
- No I/O statements have been issued.

After the following UNTANY call:

```
CALL UNTANY (IRCODE, 1, 10, IUNIT)
```

IUNIT will contain the value 1, since unit 1 is the lowest unit within the range (from 1 to 10) that is available. The file definition for the ddname FT01F001 does not affect the value returned by the UNTANY subroutine.

Example 2:

The following UNTANY call will identify the lowest unused unit, in the range from 0 to the largest unit number allowed, regardless of the file definitions in effect for the units.

```
CALL UNTANY (IRCODE, 0, -1, IUNIT)
```

XUFLOW Subroutine

The XUFLOW subroutine changes the exponent underflow mask in the program mask to allow or suppress program interrupts that could result from an exponent underflow exception.

Syntax

CALL XUFLOW (*k*)

k

is an integer expression that may have the values 0 or 1. 0 suppresses program interrupts caused by exponent underflow and the result register is set to 0. 1 allows program interrupts caused by exponent underflow to occur. The interrupt causes message AFB208I to be produced, followed by the standard or user corrective action. The standard corrective action is to set the result register to 0. Because of the time required by the operating system to handle the interrupt, and the time spent in the library to issue the message and perform the corrective action, you may notice some degradation in performance when underflows occur and interrupts are allowed.

Chapter 8. Data-in-Virtual Subroutines

VS FORTRAN Version 2 provides callable subroutines that allow you to use the Data-in-Virtual facility under MVS/XA 2.2.0 with Data Facility Product 2.3.0. Because the subroutines use character type arguments, they are supported only by VS FORTRAN Language Level 77, not Language Level 66.

For a complete description of Data-in-Virtual, see *MVS/XA Supervisor Services and Macro Instructions*, GC28-1154. Also, the technical bulletin *An Introduction to Data-in-Virtual*, GG66-0259, provides background information on using the Data-in-Virtual facility functions under MVS/XA.

Overview

Data-in-Virtual provides a specialized form of access to external data. It is similar to other means of I/O in that it is a way of making external data available to your program, but different primarily in that the actual movement of the data from external storage is deferred until your program requires it.

To use the Data-in-Virtual functions from your VS FORTRAN program, you must first create a VSAM linear data set. Information on defining VSAM linear data sets is given in *VS FORTRAN Version 2 Programming Guide*.

Then, in your VS FORTRAN program, you code the VS FORTRAN subroutines described in this chapter to map a dynamic common to all or part of the data set. You can think of the dynamic common as a "window" that enables you to "view" and make changes to the data set, which is commonly called a *data object*. (The term *data object* is preferred because you can ignore the complex record management interfaces generally associated with other types of data sets.)

The window begins at a virtual storage location and occupies a contiguous virtual address range where your VS FORTRAN program can refer to it and update it directly. Thus, you can replace complex I/O statements with ordinary references to variables.

Moreover, for applications that process large amounts of data, Data-in-Virtual can reduce the amount of virtual storage and processing time required by your program. Using other means of I/O (such as READ and WRITE statements), your program might read an entire data set into storage, process a part of the data, and write the entire data set back out to the permanent storage device. This approach can impact the availability of virtual storage if the data set is very large. Or, your program might read one record, process it, and write it out to the device. For large data sets, especially if access to the records is random or nonsequential, record processing can be very time-consuming.

In contrast, with Data-in-Virtual, only the parts of the data object that your program actually refers to are brought into virtual storage and, when you want to save changes, only those parts that your program actually changes are saved on permanent storage.

Fixed-View Versus Varying-View Subroutines

You can use one of two methods to refer to the data object: the fixed-view method or the varying-view method. With the fixed-view method, you map a single dynamic common, the window, to a given data object (see Figure 44).

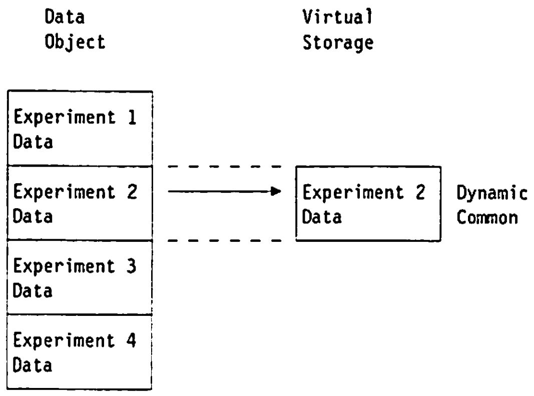


Figure 44. Fixed-View Method

With the varying-view method, you can map any number of dynamic commons to the data object and can simultaneously view different parts of it (see Figure 45 on page 295).

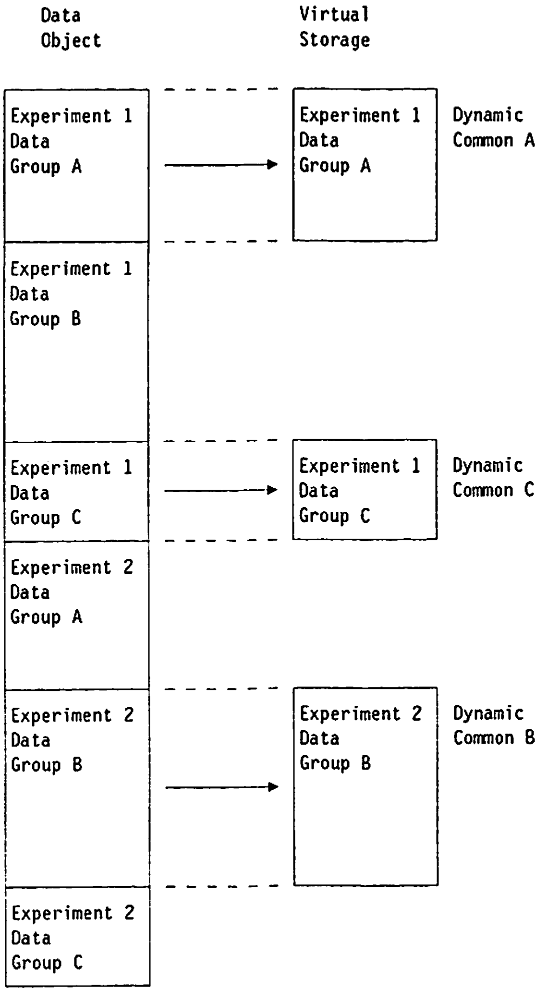

Data Object
Virtual Storage
Experiment 1 Data Group A
Experiment 1 Data Group A
Dynamic Common A
Experiment 1 Data Group B
Experiment 1 Data Group C
Experiment 1 Data Group C
Dynamic Common C
Experiment 2 Data Group A
Experiment 2 Data Group B
Experiment 2 Data Group B
Dynamic Common B
Experiment 2 Data Group C

Figure 45. Varying-View Method

For both the fixed- and varying-view methods, the basic procedure for referencing a data object is:

1. Associate the data object and access it for reading or both reading and writing.
2. Map the dynamic common to the data object.
3. View, and possibly change, the data.
4. Save the changes, if any.

At this point, you can repeat steps 3 and 4 as often as needed. You can also go back to step 2 and remap the common to another part of the data object.

5. Disassociate the common block from the data object.

It might help if you picture the data object as an "array" and the different parts of it as "array elements." Mapping a dynamic common to specific part of a data object is similar to referencing an array element. With the fixed-view method, the mappings are all the same length (because you reference each one using the same dynamic common), whereas with the varying-view method the mappings can vary in length. With the fixed-view method, you can reference one mapping at a time; with the varying-view method, you can reference any number of mappings at a time.

For both methods, each mapping that you reference begins at a page (4096 bytes) boundary because Data-in-Virtual transfers data between the data object and virtual storage in units of pages. How to take this into account when you map the dynamic common is discussed under "Sample Program with Fixed-View Subroutines" on page 307, "Sample Program with Varying-View Subroutines" on page 308, and "Ensuring Data Integrity" on page 313.

Syntax of the Subroutines

The following sections give the syntax of the Data-in-Virtual subroutines:

- DIVINF, DIVVWF, and DIVTRF, which are for the fixed-view method
- DIVINV, DIVVWV, and DIVTRV, and DIVCML which are for the varying-view method
- DIVSAV and DIVRES, which are for both the fixed- and varying-view methods

Syntax of Fixed-View Subroutines

These fixed-view subroutines give you the ability to treat a data object as one or more instances of a single dynamic common.

DIVINF Subroutine

The DIVINF subroutine allows you to associate a data object with a dynamic common for reading or for reading and writing.

Syntax

CALL DIVINF (*rcode*, *dyncom*, *objsize_commons*, *divobj*, *type*, *access*)

rcode

is a four-byte integer variable or array element that will contain the return code upon return from DIVINF.

Return Code	Explanation
0	Successful completion
4	Successful completion, but the length of the dynamic common is not an exact multiple of 4096. Unless you expect to modify the dynamic common and reuse the data object, you may ignore this return code.
8	Not a dynamic common for *dyncom*
12	Not DDNAME, DSNAME or DSN for *type*
16	Not READ or READWRITE for *access* (or blank with DDNAME type)
20	An attempt was made to access an empty data object for reading

24	An attempt was made to simultaneously access the same data object via different ddnames
28	The *divobj* specified does not refer to a VSAM linear data set
32	Invalid value specified for *divobj*
36	*Divobj* specified conflicts with *type* specified
40	Unable to dynamically allocate the VSAM linear data set specified
44	An attempt was made to use a dynamic common name already associated with another data object
48	Language level 66 CALL is not supported, an invalid parameter list was specified, or Data-in-Virtual is not supported on the operating system
128	Data-in-Virtual services failed

dyncom
is a character expression whose value is the name of a dynamic common, when trailing blanks are removed and when folded to upper case.

objsize_commons
is a four-byte integer variable or array element that will contain the size of the data object in units of the size of the dynamic common specified by *dyncom*. This value is returned by DIVINF.

divobj
is the name of the data object. It must be either a ddname or a data set name in one of the following forms:

ddname is a character expression, one to eight characters long, whose value, when trailing blanks are removed and when folded to upper case, is the ddname that identifies the VSAM linear data set. The default format of a ddname (FTnnFmmm, FTnnKkk, FTERRsss, or FTPRTsss) is not allowed.

dsn or **/dsn** is a character expression whose value is the data set name of an existing VSAM linear data set. The data set name must conform to OS naming conventions. Trailing blanks will be ignored, and lower case will be folded to upper case. The slash (/), which is not considered part of the data set name, can be used to indicate that the value specified is not a ddname.

The data set name may be specified instead of the ddname to dynamically allocate the VSAM linear data set at execution time without a DD statement. However, if the data set name is used, you must code READ or READWRITE for *access*.

type
is a character expression whose value, when any trailing blanks are removed and when folded to upper case, is DDNAME, DSNAME, or DSN to indicate the *type* of *divobj*. DDNAME indicates that the name of the VSAM linear data set is given in a DD statement. DSNAME or DSN indicates a data set name, and that the VSAM linear data set is to be dynamically allocated.

access
is a character expression whose value, when any trailing blanks are removed and when folded to upper case, is READ or READWRITE to indi-

cate the access intent. If a value of all blanks is supplied (except when the data set is to be dynamically allocated), the DISP parameter on the DD statement will control the access intent. For DISP=SHR, *access* will be READ. For DISP=OLD, *access* will be READWRITE.

Programming Notes:

A given data object may be associated with only one dynamic common at a time, and a given dynamic common may be associated with only one data object at a time. The DIVTRF subroutine must be called to disassociate a dynamic common previously associated by DIVINF.

Once the data object has been accessed through this subroutine, the DIVVWF subroutine must be called. This will establish what part of the data object is to be processed in virtual storage.

DIVVWF Subroutine

The DIVVWF subroutine establishes the part of the data object the dynamic common will map.

Syntax

CALL DIVVWF (*rcode*, *dyncom*, *mapnum*)

rcode
is a four-byte integer variable or array element that will contain the return code upon return from DIVVWF.

Return Code	Explanation
0	Successful completion
8	Not a dynamic common for *dyncom*
48	Language level 66 CALL is not supported, an invalid parameter list was specified, or Data-in-Virtual is not supported on the operating system
52	The specified range overlaps a range that is already mapped for the specified data object. The value *mapnum* should have been changed.
64	The specified dynamic common is not associated with a data object.
72	Zero or a negative value specified for *mapnum*
128	Data-in-Virtual services failed

dyncom
is a character expression whose value is the name of a dynamic common, when trailing blanks are removed and when folded to upper case.

mapnum
is an integer expression containing a number that represents the relative position of the part of the data object being mapped. The beginning of the dynamic common is mapped at that relative position in the data object.

A value of 1 indicates the beginning of the first mapping (first in *position*, not chronological order) in the data object, the value 2 indicates the second mapping, and so on. For example, for the data object shown in Figure 44 on page 294, the value 1 indicates Experiment 1 Data and the value 2 indicates Experiment 2 Data.

Each mapping must begin at a page (4096 bytes) boundary; thus, if the length of a mapping is not exactly divisible by 4096, the length is rounded up to the next page boundary. For instance, if the actual data in the first mapping is 8000 bytes, the *mapnum* value 2 indicates an offset at byte 8,192 and the value 3 indicates an offset at byte 16,384.

Programming Notes:

A given dynamic common can be mapped to only one data object at a time, and a given data object can be mapped to only one dynamic common at a time.

If you request that a mapped dynamic common be mapped again on the same data object, the first mapping will be implicitly unmapped with no changes being made to the permanent data. To make changes to the permanent data, you must explicitly save the changes by calling DIVSAV before attempting to remap the dynamic common via the DIVVWF call.

DIVTRF Subroutine

The DIVTRF subroutine terminates the association of the data object to the dynamic common.

Syntax

CALL DIVTRF (*rcode*, *dyncom*)

rcode
is a four-byte integer variable or array element that will contain the return code upon return from DIVTRF.

Return Code	Explanation
0	Successful completion
8	Invalid value specified for *dyncom*
48	Language level 66 CALL is not supported, an invalid parameter list was specified, or Data-in-Virtual is not supported on the operating system
64	The specified dynamic common is not associated with a data object.
128	Data-in-Virtual services failed

dyncom
is a character expression whose value is the name of a dynamic common, when trailing blanks are removed and when folded to upper case.

Programming Notes:

DIVTRF must be called to terminate access to a data object before the same dynamic common (*dyncom*) can be specified in another DIVINF call.

Syntax of Varying-View Subroutines

The varying-view subroutines provide you with a more flexible method for referring to a data object. These routines let you map the data object to any number of dynamic commons of possibly differing lengths.

DIVINV Subroutine

The DIVINV subroutine lets you associate a data object with a data object ID for reading or for reading and writing.

Syntax

CALL DIVINV (*rcode*, *obj-id*, *objsize_pages*, *divobj*, *type*, *access***)**

rcode

is a four-byte integer variable or array element that will contain the return code upon return from DIVINV.

Return Code	**Explanation**
0	Successful completion
12	Not DDNAME, DSNAME or DSN for *type*
16	Not READ or READWRITE for *access* (or blank with DDNAME type)
20	An attempt was made to access an empty data object for reading
24	An attempt was made to simultaneously access the same data object via different ddnames
28	The ddname specified does not refer to a VSAM linear data set
32	Invalid value specified for *divobj*
36	*Divobj* specified conflicts with *type* specified.
40	Unable to dynamically allocate the VSAM linear data set specified
48	Language level 66 CALL is not supported, an invalid parameter list was specified, or Data-in-Virtual is not supported on the operating system
128	Data-in-Virtual services failed

obj-id

is an eight-byte character variable that will contain the ID returned to identify the association between mappings and the data object.

objsize_pages

is a four-byte integer variable or array element that will contain the returned current size of the data object in units of pages (4096 bytes per page).

divobj

is the name of the data object. It must be either a ddname or a data set name in one of the following forms:

ddname
is a character expression, one to eight characters long, whose value, when trailing blanks are removed and when folded to upper case, is the ddname that identifies the VSAM linear data set. The default format of a ddname (FTnnFmmm, FTnnKkk, FTERRsss, or FTPRTsss) is not allowed.

dsn or **/dsn** is a character expression, whose value is the data set name of an existing VSAM linear data set. The data set name must conform to OS naming conventions. Trailing blanks will be ignored, and lower case will be folded to upper case. The slash (/), which is not considered part of the data set name, can be used to indicate that the value specified is not a ddname.

The data set name may be specified instead of the ddname to dynamically allocate the VSAM linear data set at execution time without a DD statement. However, if the data set name is used, READ or READWRITE must be coded for *access*.

type
is a character expression whose value, when any trailing blanks are removed and when folded to upper case, is DDNAME, DSNAME, or DSN to indicate the type of *divobj* specified. DDNAME indicates that the name of the VSAM linear data set is given in a DD statement. DSNAME or DSN indicates a data set name, and that the VSAM linear data set is to be dynamically allocated.

access
is a character expression whose value, when any trailing blanks are removed and when folded to upper case, is READ or READWRITE to indicate the access intent. If a value of all blanks is supplied, except when the data set is to be dynamically allocated, the DISP parameter on the DD statement will control the access intent. For DISP=SHR, *access* will be READ. For DISP=OLD, *access* will be READWRITE.

Programming Notes:

If you attempt a DIVINV call for a dynamic common that has access to another data object, an error results. The DIVTRV subroutine must be called before DIVINV to disassociate the dynamic common from the first data object.

Once the data object has been accessed through DIVINV, the DIVVWV subroutine must called. DIVVWV will establish what part of the data object is to be processed in virtual storage.

DIVVWV Subroutine

The DIVVWV subroutine establishes the part of the data object the dynamic common will map.

Syntax

CALL DIVVWV (*rcode*, *dyncom*, *offset*, *obj-id*)

rcode
is a four-byte integer variable or array element that will contain the return code upon return from DIVVWV.

Return Code	Explanation
0	Successful completion
4	Successful completion, but the length of the dynamic common is not an exact multiple of 4096. Unless you expect to modify the dynamic common and reuse the data object, you may ignore this return code.
8	Not a dynamic common for *dyncom*
44	An attempt was made to use a dynamic common name already associated with another data object
48	Language level 66 CALL is not supported, an invalid parameter list was specified, or Data-in-Virtual is not supported on the operating system
52	The specified range overlaps a range that is already mapped for the specified data object.
56	The *obj-id* specified is not associated with any data object.
60	Negative value specified for *offset*
128	Data-in-Virtual services failed

dyncom
is a character expression whose value, when any trailing blanks are removed and when folded to upper case, is the name of the dynamic common.

offset
is an integer expression containing the offset value in units of pages, at which the dynamic common specified by *dyncom* is to start mapping in a data object. A value of 0 indicates the beginning of the object.

The offset can be calculated using the results from the DIVCML calls. The offset is the number of pages between the beginning of the data object and the beginning of the part of the data object mapped by the window.

obj-id
is a character variable or character array element whose value is the ID returned from the DIVINV call. It is used to associate the different mappings with the data object.

Programming Notes:

A given dynamic common can be mapped to only one data object at a time.

To make changes to the data object, you must explicitly save the changes before attempting to remap the dynamic common.

Multiple calls to DIVVWV can be made on the same data object to simultaneously associate multiple dynamic commons with different parts of the data object. However, these dynamic commons may not overlap.

DIVCML Subroutine

The DIVCML subroutine obtains the length of a dynamic common. The lengths from this subroutine will provide sufficient information to enable you to map different parts of the data object, and to help you avoid overlapping mappings. You will need to keep track of the current mapping status.

Syntax

CALL DIVCML (*rcode*, *dyncom*, *length*)

rcode
is a four-byte integer variable or array element that will contain the return code upon return from DIVCML.

Return Code	Explanation
0	Successful completion
4	Successful completion, but the length of the dynamic common is not an exact multiple of 4096. Unless you expect to modify the dynamic common and reuse the data object, you may ignore this return code.
8	Not a dynamic common for *dyncom*
48	Language level 66 CALL is not supported, an invalid parameter list was specified, or Data-in-Virtual is not supported on the operating system

dyncom
is a character expression whose value, when any trailing blanks are removed and when folded to upper case, is the name of the dynamic common.

length
is a four-byte integer variable or array element that will contain the returned length of the dynamic common in pages (4096 bytes per page). For example, if the dynamic common is 5000 bytes long, the *length* returned is 2.

Programming Note:

For an example of how *length* can be used to calculate the *offset* value for the DIVVWV subroutine, see the sample programs in Figure 48 on page 309 and Figure 49 on page 311.

DIVTRV Subroutine

The DIVTRV subroutine terminates the association between the data object ID and the data object.

Syntax

CALL DIVTRV (*rcode*, *obj-id*)

rcode

is a four-byte integer variable or array element that will contain the return code upon return from DIVTRV.

Return Code	Explanation
0	Successful completion
48	Language level 66 CALL is not supported, an invalid parameter list was specified, or Data-in-Virtual is not supported on the operating system
56	Invalid value specified for *obj-id*
128	Data-in-Virtual services failed

obj-id

is a character variable or array element of at least eight bytes long which contains the ID value returned by DIVINV for the data object.

Programming Notes:

DIVTRV must be called to terminate access to a data object before it can be specified in another DIVINV call.

All dynamic commons that are mapped to a data object when a DIVTRV call is made will automatically be unmapped.

Syntax of Common Subroutines

The DIVSAV and DIVRES subroutines can be used with both the fixed-view or varying-view methods.

DIVSAV Subroutine

The DIVSAV subroutine saves changes made in the dynamic common to the data object that has been accessed for READWRITE.

Syntax

CALL DIVSAV (*rcode*, *dyncom***)**

rcode
is a four-byte integer variable or array element that will contain the return code upon return from DIVSAV.

Return Code	Explanation
0	Successful completion
8	Invalid value specified for *dyncom*
48	Language level 66 CALL is not supported, an invalid parameter list was specified, or Data-in-Virtual is not supported on the operating system
64	The specified dynamic common is not associated with a data object.
68	DIVSAV requested, but the specified data object is not accessed in READWRITE mode.
128	Data-in-Virtual services failed

dyncom
is a character expression whose value is the name of a dynamic common, when trailing blanks are removed and when folded to upper case.

DIVRES Subroutine

The DIVRES subroutine resets the data in the dynamic common to the values in the mapped part of the data object, eliminating any changes that have been made in the dynamic common, either initially or since the last DIVSAV.

Syntax

CALL DIVRES (*rcode*, *dyncom***)**

rcode
is a four-byte integer variable or array element that will contain the return code upon return from DIVRES.

Return Code	Explanation
0	Successful completion
8	Invalid value specified for *dyncom*
48	Language level 66 CALL is not supported, an invalid parameter list was specified, or Data-in-Virtual is not supported on the operating system
64	The specified dynamic common is not associated with a data object.
128	Data-in-Virtual services failed

dyncom
is a character expression whose value is the name of a dynamic common, when trailing blanks are removed and when folded to upper case.

Interface to the Data-in-Virtual Functions

Figure 46 shows the Data-in-Virtual functions to which the subroutines for the fixed- and varying-view subroutines interface. The Data-in-Virtual functions are described in *MVS/XA Supervisor Services and Macro Instructions*, GC28-1154.

FIXED-VIEW SUBROUTINES	VARYING-VIEW SUBROUTINES	DESCRIPTION	DIV FUNCTION
DIVINF	DIVINV	Associates the data object with a dynamic common and indicates whether the data object is to be accessed for reading or for both reading and writing.	IDENTIFY and ACCESS
DIVVWF	DIVVWV	Establishes the relative location in the data object where the dynamic common is to be mapped. In the case of remapping, also unmaps the previous mapping.	MAP and UNMAP
DIVSAV	DIVSAV	Saves changes made in the dynamic common to the data object.	SAVE
DIVRES	DIVRES	Resets the dynamic common to the data contained in the data object, discarding any changes.	RESET
DIVTRF	DIVTRV	For the fixed-view method, terminates the association between the data object and the dynamic common. For the varying-view method, terminates the association between the data object and the data object ID.	UNMAP, UNACCESS, and UNIDENTIFY
None	DIVCML	Obtains the length of a dynamic common.	None

Figure 46. Data-in-Virtual Subroutines

Sample Program with Fixed-View Subroutines

The use of the fixed-view subroutines is shown in Figure 47. The figure illustrates the following basic process:

- The data object is accessed for update (DIVINF).
- The dynamic common is mapped to part of the data object (DIVVWF).
- The window contents are saved into the previously empty data object (DIVSAV).
- The dynamic common is unmapped and is remapped to another part of the data object (DIVVWF).
- The window contents are saved into the data object (DIVSAV).
- Access to the data object is terminated (DIVTRF).

Note that to indicate where in the data object you want to map the dynamic common, you simply specify, with the *mapnum* parameter on the DIVVWF subroutine, the relative position of the area. For example, for the data object shown in Figure 44 on page 294, the value 1 indicates Experiment 1 Data, the value 2 indicates Experiment 2 Data, and so on.

If the dynamic common size is not an integral number of pages in size, a return code of 4 will be returned by DIVINF.

```
@PROCESS DC(DIVDC1)
C
C Example of Fixed-View Subroutines
C
C The data object (accessed via the ddname 'DIVOBJ') contains one
C style of data organization, described by the dynamic common DIVDC1.
C
      Program Example
      Common /DIVDC1/ MyArray
      Integer*4 RC/0/, MaxInst/0/
      Real*8 MyArray(512)
      Character*8 Divdd/'DIVOBJ'/, Comnam/'DIVDC1'/

C Assign some values to the array, then write them to the data object.
C The array is 4096 bytes long.

      Print *, MyArray(1), MyArray(256), MyArray(512)
      Call DIVINF (RC, Comnam, MaxInst, Divdd, 'DDNAME', 'READWRITE')

C Check the return code.

      If (RC .NE. 0) Call Error(RC, 'DIVINF')

C Map the dynamic common to the first mapping of the data object.

      Call DIVVWF (RC, Comnam, 1)

C Check the return code.

      If (RC .NE. 0) Call Error(RC, 'DIVVWF')

C Assign values to the array.

      MyArray(  1) = 1.0D0
      MyArray(256) = 2.0D0
      MyArray(512) = 3.0D0
```

Figure 47 (Part 1 of 2). Example 1 of Data-in-Virtual Subroutines: Fixed-View Method

```
C Save the window contents into the object.

      Call DIVSAV (RC, Comnam)

C Check the return code.

      If (RC .NE. 0) Call Error(RC, 'DIVSAV')

      Print *, MaxInst, MyArray(1), MyArray(256), MyArray(512)

C Map the dynamic common to the second mapping of the data object.

      Call DIVVWF (RC, Comnam, 2)

C Check the return code.

      If (RC .NE. 0) Call Error(RC, 'DIVVWF')

C Assign values to the array.

      MyArray(  1) = 4.0D0
      MyArray(256) = 5.0D0
      MyArray(512) = 6.0D0

C Save the window contents into the object.

      Call DIVSAV (RC, Comnam)

C Check the return code.

      If (RC .NE. 0) Call Error(RC, 'DIVSAV')

      Print *, MaxInst, MyArray(1), MyArray(256), MyArray(512)

C We do not plan to do any more processing on this object.

      Call DIVTRF (RC, Comnam)

C Check the return code.

      If (RC .NE. 0) Call Error(RC, 'DIVTRF')

      End
@PROCESS
      Subroutine Error(Code, Rtn)

C Print a message if a DIV subroutine Call was not successful.

      Integer*4  Code
      Character*(*) Rtn
      Write (6, 1) Rtn, Code
    1 Format(' Routine ', A, ' returned non-zero code ', I4)
      Stop
      End
```

Figure 47 (Part 2 of 2). Example 1 of Data-in-Virtual Subroutines: Fixed-View Method

Sample Program with Varying-View Subroutines

The varying-view subroutines let you map any number of dynamic commons to different parts of the data object, as long as the parts do not overlap. Each dynamic common can be mapped to only one part of the data object at a time; if you try to simultaneously map it to another part, the first part is unmapped, discarding any changes that you haven't saved.

This method is more complex than the fixed-view method and it does require you to provide and maintain more detailed information. Unlike the fixed-view method, which has a one-to-one association between a dynamic common and a data object, the varying-view method permits multiple dynamic commons. Therefore, a parameter on the DIVINV subroutine is provided for assigning an identifying token to the data object. You then refer to this token on the call to the mapping routine, DIVVWV, for each dynamic common.

Because the dynamic commons may vary in length, mapping them to the data object also requires more user control than with the fixed-view method because you do the storage management. On the DIVVWV call, you must indicate the offset, in units of pages, from the beginning of the data object to the beginning of the part of the object to be made visible in the window. You can calculate this offset with the length returned by the DIVCML subroutine. The DIVCML subroutine will give you the length, in units of pages, of a dynamic common.

The *objsize_pages* value obtained from the DIVINV call indicates how much of the data object has been used at the time of the call. With this information, you know the next available offset for new output, as well as how far you can go with previously stored data. This, in effect, is your end-of-data indication.

Figure 48 and Figure 49 on page 311 illustrate the use of the varying-view subroutines.

```
@PROCESS DC(COM_A,COM_B,COM_C)

C Example of simple use of Varying-View Subroutines

C The data object (accessed via the ddname 'MYOBJECT') contains three
C   styles of data organization. The styles are described by the
C   three dynamic commons COM_A, COM_B, and COM_C.

C   In this example, there is only a  single occurrence in the data
C   object of each type of mapped data. Note that no checking of
C   return codes is performed.  In an actual program, this checking
C   should be performed.

      Common / COM_A / NAD, ADATA(5000)
      Common / COM_B / NBD, NBI, BITEM(200), BDATA(2000)
      Common / COM_C / NCD, CTEMP(40,75), CDATA(8000)

      Character*8 Token
      Integer Length_A, Length_B, Length_C
      Integer Offset_A, Offset_B, Offset_C, TotCom, Size

C -----------------------------------------------------------------
C Determine the length of each dynamic common, to determine the
C   offset into the data object of the data each one maps.

      Call DIVCML(IRet, 'COM_A', Length_A)
      Call DIVCML(IRet, 'COM_B', Length_B)
      Call DIVCML(IRet, 'COM_C', Length_C)

        Offset_A = 0
        Offset_B = Offset_A + Length_A
        Offset_C = Offset_B + Length_B
      TotCom = Offset_C + Length_C
```

Figure 48 (Part 1 of 2). Example 2 of Data-in-Virtual Subroutines: Varying-View Method

```
C Obtain access to the DIV data object.

      Call DIVINV(IRet, Token, Size, 'MYOBJECT', 'DDNAME', 'READWRITE')

C Assume we don't want to extend the size of the object.

      IF (TOTCOM .GT. Size) THEN
         Call DIVTRV(IRet, Token)
         Stop 'Object size is too small.'
      ENDIF

C Now, provide a view for each dynamic common onto its associated
C   portion of the data object.

      Call DIVVWV(IRet, 'COM_A', Offset_A, Token)
      Call DIVVWV(IRet, 'COM_B', Offset_B, Token)
      Call DIVVWV(IRet, 'COM_C', Offset_C, Token)

C Now, call subroutine WORK to use the data in the dynamic commons.

      Call WORK

C Save changed data in the dynamic commons back into the data object.

      Call DIVSAV(IRet, 'COM_A')
      Call DIVSAV(IRet, 'COM_B')
      Call DIVSAV(IRet, 'COM_C')

C Terminate the associations with the data object.

      Call DIVTRV(IRet, Token)
      End
```

Figure 48 (Part 2 of 2). Example 2 of Data-in-Virtual Subroutines: Varying-View Method

```
@PROCESS DC(COM_A,COM_B,COM_C)

C Example of more elaborate use of Varying-View Subroutines

C The data object (accessed via the ddname 'MYOBJECT') contains three
C   styles of data organization. The styles are described by the
C   three dynamic commons COM_A, COM_B, and COM_C.

C   In this example, there are multiple occurrences in the data
C   object of each type of mapped data, in repeating groups of
C   (COM_A, COM_B, COM_C).
C   Note that insufficient checking of return codes is performed.

      Common / COM_A / NAD, ADATA(5000)
      Common / COM_B / NBD, NBI, BITEM(200), BDATA(2000)
      Common / COM_C / NCD, CTEMP(40,75), CDATA(8000)

      Character*8 Token
      Integer Length_A, Length_B, Length_C
      Integer Offset_A, Offset_B, Offset_C, TotCom, Size
      Integer Inst_A, Inst_B, Inst_C

C -----------------------------------------------------------------
C Determine the length of each dynamic common, to determine the
C   offset into the data object of the data each one maps.

      Call DIVCML(IRet, 'COM_A', Length_A)
      Call DIVCML(IRet, 'COM_B', Length_B)
      Call DIVCML(IRet, 'COM_C', Length_C)

        Offset_A = 0
        Offset_B = Offset_A + Length_A
        Offset_C = Offset_B + Length_B
      TotCom = Offset_C + Length_C

C Obtain access to the DIV data object.

      Call DIVINV(IRet, Token, Size, 'MYOBJECT', 'DDNAME', 'READWRITE')
      IF (IRet .NE. 0) THEN
         Print *, 'Return code is ',IRet
         Stop 1
      ENDIF

C Suppose we have determined that we want to view the 5th instance of
C   COM_A, the 17th instance of COM_B, and the 41st instance of COM_C.

      Inst_A =  5
      Inst_B = 17
      Inst_C = 41

C Now, provide a view for each dynamic common onto its associated
C   portion of the data object. Calculate the needed offsets.

  100 ITemp = (Inst_A - 1) * TotCom + Offset_A
      Call DIVVWV(IRet, 'COM_A', ITemp, Token)

      ITemp = (Inst_B - 1) * TotCom + Offset_B
      Call DIVVWV(IRet, 'COM_B', ITemp, Token)

      ITemp = (Inst_C - 1) * TotCom + Offset_C
      Call DIVVWV(IRet, 'COM_C', ITemp, Token)
```

Figure 49 (Part 1 of 2). Example 3 of Data-in-Virtual Subroutines: Varying-View Method

```
C Now, call subroutine WORK to use the data in the dynamic commons.

      Call WORK

C Save changed data in the dynamic commons back into the data object.

      Call DIVSAV(IRet, 'COM_A')
      Call DIVSAV(IRet, 'COM_B')
      Call DIVSAV(IRet, 'COM_C')

C At this point, we would normally choose new values for Inst_A,
C   Inst_B, and Inst_C, and loop back to statement 100 to establish
C   the new views.

C Terminate the associations with the data object.

      Call DIVTRV(IRet, Token)
      End
```

Figure 49 (Part 2 of 2). Example 3 of Data-in-Virtual Subroutines: Varying-View Method

Remapping a Dynamic Common to Different Parts of the Data Object

With either the fixed-view method or the varying-view method, you can remap a dynamic common to view and update different parts of the data object. In other words, you can map the dynamic common to one part of the data object, process the data and save your changes, then remap the dynamic common to another part of the data object to process more data.

In the fixed-view example shown in Figure 50, different parts of the data object, mapped by the dynamic common /USAGE/, are accessed sequentially by varying the parameter K (the *mapnum* parameter).

```
@PROCESS DC(USAGE)
      Subroutine SUB3
      Common /USAGE/ A(500), M(500)
      Real A
      Integer K, RetCod
      .
      .
      .
      Call DIVINF(RetCod, 'USAGE', NOccur,
     X          'DIVOBJ', 'DDNAME', 'READ')
      If (RetCod .GT. 4) Stop 'Access Failed'
      Do 8 K = 1, NOccur
        Call DIVVWF(RetCod, 'USAGE', K)
        If (RetCod .NE. 0) Stop 'Viewing Unavailable'
        --- process the data in dynamic common USAGE ---
    8 Continue
      ...
      End
```

Figure 50. Example of Remapping: Fixed-View Method

Resetting a Dynamic Common

Based on processing performed by your program, you may determine that the data in the window is not what you intended. You can use the DIVRES subroutine to reset the window with data in the data object since the last DIVSAV.

Ensuring Data Integrity

Because of the way that Data-in-Virtual accesses a data object, it is strongly recommended that you explore potential impacts to your application program when you use different mappings of commons.

Data-in-Virtual requires that all references and accesses to data in data objects be managed in units of pages (4096 bytes per page), starting on an address boundary evenly divisible by 4096. MVS provides this management of each independent common. So when you establish a dynamic common that has a length that is not an exact multiple of 4096, there is some unused space in the last page at the end of the common. This unused space will appear in the data written to your data object in the form of gaps of undefined data at the end of the last page.

Dependence on the page organization of Data-in-Virtual will not be a problem for your application if you use the same dynamic commons when you first write data to the data object and every time you access it later. To facilitate this, you can place the COMMON specification statements in INCLUDE files.

Also, keep in mind that data in the dynamic common window is lost anytime the window is reset, remapped, or terminated; therefore, it is a good practice to avoid using the window to pass arguments to Data-in-Virtual subroutines.

Performance and Storage Factors

In evaluating the various organization and mapping alternatives, you will want to consider the available virtual storage and the performance measurements of your application as it runs in your environment.

For example, whether you view an entire entire data object all at once or view parts of it separately will likely depend on the amount of virtual storage available.

Note that because dynamic commons are allocated in units of pages (4096 bytes per page), using dynamic commons much smaller than 4096 bytes can result in much unused space.

Will your program's data access patterns be rapid and random? If so, you might decide to create a single large common to accommodate more of the data in storage. This way, you will have to do little or no remapping to access the data. On the other hand, if all parts of the data can be accessed independently of each other and a fairly large amount of computation is performed before the next part is required, you may decide to view only one part at a time with a single dynamic common, or else view each part simultaneously with different dynamic commons.

Effect on Optimization

Because the Data-in-Virtual functions are requested through the call interface, optimization by the VS FORTRAN compiler might be impacted. This is because loops with these Data-in-Virtual calls are not as fully optimizable. Loops with Data-in-Virtual calls are also not vectorizable.

You should keep Data-in-Virtual calls outside of the inner loops as much as possible in order to obtain the highest possibility of optimized code when using OPT(2), OPT(3), or VECTOR.

Using Data-in-Virtual in an MTF Environment

If you are running your applications in a Multitasking Facility (MTF) environment, you must make all Data-in-Virtual subroutine calls from your main task program. However, a dynamic common that is shared by the main task program and the parallel subroutines may be used to supply access to the data object for all of the routines.

When sharing a dynamic common among several parallel subroutines, do not have the main task reset, terminate, or remap the dynamic common until all subroutines have completed. Otherwise, the results will be unpredictable. See *VS FORTRAN Version 2 Programming Guide* for information on sharing dynamic commons under MTF.

Chapter 9. Extended Error-Handling Subroutines and Error Option Table

Extended Error Handling

Five subroutines are provided by VS FORTRAN Version 2 for use in extended error handling: ERRMON, ERRSAV, ERRSET, ERRSTR, and ERRTRA. These subroutines enable you to alter the error option table dynamically. Existing error conditions can be changed and user exits can be supplied.

The error option table is a list of errors detected by the VS FORTRAN Version 2 library. Each error is represented by an entry in the option table, which contains values for information related to the error. The option table (as shipped in the library, module AFBUOPT) is preset with information for IBM-designated error conditions.

Changes made dynamically to the option table, using the error-handling subroutines, are in effect for the duration of the program that made the change. That is, the copy of the option table in the executing program is changed, but the copy in the library remains unchanged.

The option table is generated by the macro VSF2UOPT, and the macro may be used to customize the error option table for your environment. Your system administrator will know whether and how the error option table has been modified for your environment.

The syntax for each of the error-handling subroutines is shown below, under "Error-Handling Subroutines." The details of the error option table and its preset information are given under "Error Option Table" on page 322. For an explanation of how to use extended error handling, see *VS FORTRAN Version 2 Programming Guide.*

Error-Handling Subroutines

ERRMON Subroutine

The ERRMON subroutine calls the error monitor routine, the same routine used by VS FORTRAN Version 2 when it detects an error.

Syntax

CALL ERRMON (*imes, iretcd, ierno* **[,***data1* **] [,***data2* **...])**

imes

The name of an array, aligned on a fullword boundary, that contains, in EBCDIC characters, the text of the message. The number of the error condition should be included as part of the text, because the error monitor prints **only** the text passed to it. The first element of the array contains an integer whose value is the length of the message. Thus, the first 4 bytes of the array are not printed. *imes* must point to the integer * 4 length. Immediately following that is the text of the message in EBCDIC characters. If the message length is greater than the record length of the error message unit, the message is printed on two or more lines of printed output.

iretcd

An integer variable made available to the error monitor for setting the following return codes:

0 The option table or user-exit routine indicates that standard correction is required.

1 The option table indicates that a user exit to a corrective routine has been executed. The function is to be reevaluated using arguments supplied in the parameters: *data1,data2*...

For input/output type errors, the value 1 indicates that standard correction is not wanted.

ierno

The error condition number in the option table. If any number specified is not within range of the option table, an error message is printed.

data1,data2...

Variable names in an error-detecting routine for the passing of arguments found to be in error. One variable must be specified for each argument. Upon return to the error-detecting routine, results obtained from corrective action are in these variables. Literals and variables which you do not want altered should not be in a CALL statement because there they are subject to change.

Because *data1* and *data2* are the parameters that the error monitor passes to a user-written routine to correct the detected error, care must be taken to make sure that these parameters agree in type and number in a call to ERRMON and/or in a call to a user-written corrective routine, if one exists.

ERRMON examines the option table for the appropriate error number and its associated entry and takes the actions specified. If a user-exit address has been specified, ERRMON transfers control to the user-written routine indicated by that address. Thus, the user has the option of handling errors in one of two ways:

- Call ERRMON without supplying a user-written exit routine.
- Call ERRMON and providing a user-written exit routine.

Example:

```
CALL ERRMON (MYMSG,ICODE,315,D1,D2)
```

The example states that the message to be printed is contained in an array named MYMSG; the field named ICODE is to contain the return code; the error condition number to be investigated is 315; and arguments to be passed to the user-written routine are contained in fields named D1 and D2.

ERRSAV Subroutine

The ERRSAV subroutine copies an option table entry into an 8-byte storage area accessible to the FORTRAN programmer.

Syntax

CALL ERRSAV (*ierno*, *tabent***)**

ierno
The error number in the option table. Should any number not within the range of the option table be used, an error message will be printed.

tabent
The name of an 8-byte storage area in which the option table entry is to be stored.

Example:

```
CALL ERRSAV (213,ALTERX)
```

The example states that the entry for error number 213 is to be stored in the area named ALTERX.

ERRSET Subroutine

The ERRSET subroutine permits the user to control execution when error conditions occur. For a range of error messages, the user can specify:

- How many errors can occur before execution ends
- How many error messages can be printed
- Whether a traceback is to be printed
- Whether a user exit routine is to be executed

Syntax

CALL ERRSET (*ierno*, *inoal* **[**, *inomes* **] [**, *itrace* **] [**, *iusadr* **] [**, *irange* **]**)

ierno

The error number. Should any number not within the range of the option table be used, an error message will be printed. (If *ierno* is specified as 212, there is a special relationship between the *ierno* and *irange* parameters. See the explanation of *irange*, following.)

inoal

An integer specifying the number of errors permitted before each execution is terminated. If *inoal* is specified as either 0 or a negative number, the specification is ignored, and the number-of-errors option is not altered. If a value of more than 255 is specified, an unlimited number of errors is permitted.

The value of *inoal* should be set at 2 or greater if transfer of control to a user-supplied error routine is desired after an error. If this parameter is specified with a value of 1, execution is terminated after only one error.

inomes

An integer indicating the number of messages to be printed. A negative value specified for *inomes* suppresses all messages; a specification of zero indicates that the number-of-messages option is not to be altered. If a value greater than 255 is specified, an unlimited number of error messages is permitted.

itrace

An integer whose value may be 0, 1, or 2. A specification of 0 indicates the option is not to be changed; a specification of 1 requests that no traceback be printed after an error; a specification of 2 requests a traceback be printed after each error occurrence. (If a value other 1 or 2 is specified, the option remains unchanged.)

iusadr

Specifies one of the following:

- The value 1, indicating that the option table is to be set to show no user-exit routine (that is, standard corrective action is to be used when continuing execution).
- The name of a closed subroutine that is to be executed after the occurrence of the error identified by *ierno*. The name must appear in an EXTERNAL statement in the source program, and the routine to which control is to be passed must be available at link-editing time.

- The value 0, indicating that the table entry is not to be altered.

See "Coding the User Exit Routine," below.

irange
An error number higher than that specified in *ierno*. This number indicates that the options specified for the other parameters are to be applied to the entire range of error conditions encompassed by *ierno* and *irange*. (If *irange* specifies a number lower than *ierno*, the parameter is ignored.)

If this parameter is omitted, only the options for the single error number *ierno* are applied.

Example:

```
CALL ERRSET (310,20,5,0,MYERR,320)
```

This example specifies the following:

- Error condition 310 (*ierno*).
- The error condition may occur up to 20 times (*inoal*).
- The corresponding error message may be printed up to 5 times (*inomes*).
- The current action for traceback information is to remain in force (*itrace*).
- The user-written routine MYERR is to be executed after each error (*iusadr*).
- The same options are to apply to all error conditions from 310 to 320 (*irange*).

Coding the User Exit Routine

When a user exit routine address is supplied in the option table entry for a given error number, the error monitor calls the specified subroutine for corrective action. The subroutine is called by assembler language code equivalent to the following statement:

```
CALL x (iretcd,ierno,data1,data2...)
```

where *x* is the name of the routine whose address was placed in the option table by the *iusadr* parameter of the CALL ERRSET statement. The parameters *iretcd*, *ierno*, *data1*, *data2*... correspond to the parameters specified for each error message in Figure 53 on page 326 through Figure 55 on page 331.

If an error occurs during input/output, subroutine *x* must not execute any FORTRAN I/O statements, for example, OPEN, CLOSE, INQUIRE, READ, WRITE, BACKSPACE, ENDFILE, REWIND, REWRITE, DEBUG, or PAUSE, or any calls to PDUMP or ERRTRA. Subroutine *x* must not call the library routine that detected the error, or any routine that uses that library routine. For example, a statement such as:

```
R = A**B
```

cannot be used in the exit routine for error number 252, because the library subroutine FRXPR# uses EXP, which detects error number 252.

Standard or user-supplied corrective action is indicated by setting the return code (*iretcd*), as follows:

1. If *iretcd* is set to 0, standard corrective action is requested; *data1* and *data2* must not be altered by the routine. If *data1* and *data2* are altered when *iretcd* is set to 0, the operations that follow will have unpredictable results.
2. If *iretcd* is set to 1, the execution-time library reacts to the user-supplied correction action specified in Figure 53 on page 326 through Figure 55 on page 331.
3. Only the values 0 and 1 are valid for *iretcd*. A user exit routine must ensure that one of these values is used if it changes the return code setting.

The user-written exit routine can be written in FORTRAN or in assembler language. In either case, it must be able to accept the call to it as shown above. The user exit routine must be a closed subroutine that returns control to the caller. Caution should be used when changing the values of any variables in the common area while in a closed user error-handling routine under optimization level 1, 2, or 3. Certain control flow and variable usage information will not be known to the optimizer.

If the user-written exit routine is written in assembler language, the end of the parameter list can be checked. The high-order bit of the last parameter will be set to 1. Standard register linkage conventions are followed, using registers 13, 14, 15, and 1.

If the routine is written in FORTRAN, the parameter list must match in length the parameter list passed in the CALL statement issued to the error monitor.

Actions that users may take if they want to correct an error are described in Figure 53 on page 326 through Figure 55 on page 331.

ERRSTR Subroutine

The ERRSTR subroutine stores an entry in the option table.

Syntax

CALL ERRSTR (*ierno*, *tabent*)

ierno
The error number for which the entry is to be stored in the option table. Should any number not within the range of the option table be used, an error will be printed.

tabent
The name of an 8-byte storage area containing the table entry data.

Example:

```
CALL ERRSTR (213,ALTREX)
```

The example states that an entry for error number 213, stored in ALTREX, is to be restored to the option table.

ERRTRA Subroutine

The ERRTRA subroutine dynamically requests a traceback and continued execution.

Syntax

CALL ERRTRA

The CALL ERRTRA statement has no parameters.

Error Option Table

The structure of option table entries is shown in Figure 51. Figure 52 on page 324 lists the preset information for each error condition. Figure 52 through Figure 55 on page 331 summarize the preset information for standard or user-supplied corrective action. The preset entries that cannot be altered are identified in Figure 52.

Field Contents	Field Length	Default Value[1]			Field Description
Number of error occurrences	1 byte	10 [2]			Number of times this error condition should be allowed to occur. When the value of the error count field (below) equals this value, job processing is terminated. Number may range from 0 to 255. A value of 0 means an unlimited number of occurrences. [3]
Number messages to print	1 byte	5 [4]			Number of times the corresponding message is to be printed before message printing is suppressed. A value of 0 means no message is to be printed.
Error count	1 byte	0			The number of times this error has occurred. A value of 0 indicates that no occurrences have been encountered.
Option bits	1 byte	42 (hex)			Eight option bits defined as follows (the default setting has an asterisk):
			Bit	**Setting**	**Explanation**
			0	0*	Error condition is not an I/O error.
				1	Error condition is an I/O error. Occurrences are not to be counted if ERR or IOSTAT parameter is given.
			1	0	Table entry cannot be modified. [5]
				1*	Table entry can be modified.
			2	0*	Fewer than 256 errors have occurred.
				1	More than 256 errors have occurred. (Add 256 to error count field above to determine the number.)
			3[6]	0*	Do not print buffer with error messages.
				1	Print buffer contents.
			4	0*	Reserved.
			5	0*	Print messages default number of times only.
				1	Unlimited printing requested; print for every occurrence of error.
			6[7]	0	Do not print traceback map.
				1*	Print traceback map.
			7	0*	Reserved.
User exit	4 bytes	1			Indicates where a user corrective routine is available. A value other than 1 specifies the address of the user-written routine.

Figure 51. Error Option Table Entry

Error Code	Parameters Passed To User Exit	Standard Corrective Action	User-Supplied Corrective Action
214	A,B,D	*Input*: Ignore remainder of I/O list. Ignore input/output request if for ASCII tape. *Output*: If unformatted write initially requested, change record format to VS. If formatted write initially requested, ignore input/output request.	If user correction is requested, the remainder of the I/O list is ignored.
215	A,B,E	Substitute zero for the invalid character.	The character placed in E will be substituted for the invalid character; input/output operations may not be performed. See Note 3.
217	A,B,D	Increase sequence number and read next file.	Note 1
218	A,B,D,F	Ignore remainder of I/O list.	Note 1
219-224	A,B,D	Ignore remainder of I/O list.	Note 1
225	A,B,E	Substitute 0 for the invalid character.	The character placed in E will be substituted for the invalid character. See Note 3.
226	A,B,R	R=0 for input number too small. R=16**63 - 1 for input number too large.	User may alter R.
227	A,B,D	Ignore remainder of I/O list.	Note 1
231	A,B,D	Ignore remainder of I/O list.	Note 1
232	A,B,D,G	Ignore remainder of I/O list.	Note 1
233	A,B,D	Change record number to list maximum allowed (32000).	Note 1
234, 236	A,B,D	Ignore remainder of I/O list.	Note 1
237	A,B,D,F	Ignore remainder of I/O list.	Note 1
238	A,B,D	Ignore remainder of I/O list.	Note 1
240	Note 4	Program termination	None
286	A,B,D	Ignore request	Note 1
287	A,B,D	Ignore request	Note 1
288	A,B,D	Implied wait	Note 1

Figure 53 (Part 2 of 2). Corrective Action after Error

Notes to Figure 53:

The alphabetic characters used in the "Parameters Passed to User" column have the following meanings:

Parameter	Meaning
A	Address of return code field (INTEGER*4)
B	Address of error number (INTEGER*4)
C	Address of invalid format character (see Note 5)
D	Address of data set reference number (INTEGER*4)
E	Address of invalid character (see Note 5)
F	Address of DECB
G	Address of record number requested (INTEGER*4)
I	Result after conversion (INTEGER*4)
J	Address of value of key argument
K	Address of length of key argument supplied
L	Address of KEYID value
M	Address of beginning of record
N	Address of length of record
O	Address of VSAM return code
P	Address of error or feedback code
Q	Address of key in record previously read
R	Result after conversion (REAL*4)

[1] If the error was not caused during asynchronous input/output processing, the user exit-routine cannot perform any asynchronous I/O operation and, in addition, may not perform any REWIND, BACKSPACE, or ENDFILE operation. If the error was caused during asynchronous input/output processing, the user cannot perform any input/output operation. On return to the library, the remainder of the input/output request will be ignored.

If error condition 218 (input/output error detected) occurs while error messages are being written to the object error data set, the message is written to the console and the job is terminated. If no DD card has been supplied for the object error data set, error message AFB219I is written out at the console and the job is terminated.

[2] The user exit routine may supply an alternative answer for the setting of the result register. The routine should always set an INTEGER*4 variable and the VS FORTRAN Version 2 library will load fullword or halfword depending on the length of the argument causing the error.

[3] Alternatively, the user can set the return code to 0, thus requesting a standard corrective action.

[4] Code 240 generates a message showing the system or program code causing program termination, the address of the STAE control block, and the contents of the last PSW when abnormal termination occurred. Further information appears under message code AFB240 in Appendix D, "Library Procedures and Messages" on page 375.

[5]

If LANGLVL(66), then LOGICAL*1.

If LANGLVL(77), then CHARACTER*1.

Error Code	Parameters Passed to User Exit [1]	Reason for Interrupt [2]	Standard Corrective Action	User Supplied Corrective Action
207	A,B,D,I	Overflow Integer overflow (Interrupt code 8) Exponent overflow (Interrupt code C)[4]	For exponent overflow, the result register is set to the largest floating-point number. The sign of the result register is not altered. No standard fixup for other interrupts.	For exponent overflow, the user may alter D[3].
208	A,B,D,I	Underflow Exponent underflow (Interrupt code D).	The result register is set to zero.	The user may alter D[3].
209	A,B,D,I	Divide check Integer divide (Interrupt code 9) Decimal divide (Interrupt code B) Floating-point divide (Interrupt code F)[4]	For floating-point divide, where N/0 and N=0, the result register is set to 0. Where N ¬ 0, the result register is set to the largest possible floating point number. No standard fixup for other interrupts.	For floating-point divide, the user may alter D[3].
210	A,B	Operation exception (Interrupt code 1) Specification exception (Interrupt code 6) Data exception (Interrupt code 7)	No special corrective action other than correcting boundary misalignment for some specification exceptions.	Note[5]

Figure 54. Corrective Action after Program Interrupt

Notes to Figure 54:

[1] The variable types and meanings are as follows:

Variable	Type	Meaning[6]
A	INTEGER*4	The return code field.
B	INTEGER*4	The error number.
D	REAL*16	The result register after the interrupt.
I	INTEGER*4	The exponent is an integer value for the number in D. The value in I is not the true exponent, but what was left in the exponent field of the floating-point number after the interrupt.

[2] Program interrupts are described in the appropriate Principles of Operations publication, listed in the preface.

[3] The user exit routine may supply an alternative answer for the setting of the result register. This is accomplished by replacing the value in D. Although the interrupt may be caused by a short, long, or extended floating-point operation, the user exit routine need not be concerned with this. The user exit routine should always set the correct length, but may set a REAL*16 variable and the VS FORTRAN Version 2 library will load the correct length data item, depending on the floating-point operation that caused the interrupt. For interrupts other than floating point, the user exit routine does not have the ability to change the result register and any data placed in D is ignored.

[4] For floating-point interrupts, the result register is shown in the message. For interrupts other than floating point, the result register contains zeros.

[5] The boundary alignment adjustments are informative messages; there is nothing to alter before execution continues.

[6] These are returned in a parameter list.

If a program is going to use them, the SUBROUTINE statement may be specified as:

```
SUBROUTINE HYEXIT(IRC,IERR,DREG,IEXP)
```

where IRC, IERR, DREG, and IEXP correspond to A, B, D, and I, respectively, and DREG is given a type of REAL*16.

If an assembler language program is going to use them, they are pointed to by register 1 in the standard OS/VS convention of a list of addresses, each of which points to A, B, D, and I.

Error Code	FORTRAN Reference[1]	Invalid Argument Range	Options Standard Corrective Action[2], [3]	Options Parameters Passed to User Exit[4]
118	XA=X**Y	X < 0, Y ≠0	XA=$\lvert X\rvert$**Y	A, B, X, Y
119	DA=D**DB	D < 0, DB ≠ 0	DA=$\lvert D\rvert$**DB	A, B, D, DB
241	K=I**J	I=0, J≤0	K=0	A, B, I, J
242 [5]	Y=X**I	X=0, I≤0	If I=0, Y=1 If I < 0, Y=•	A, B, X, I
243 [5]	DA=D**I	D=0, I≤0	If I=0, Y=1 IF I < 0, Y=•	A, B, D, I
244	XA=X**Y	X=0, Y≤0	If Y=0, XA=1 If Y<0, XA=•	A, B, X, Y
245	DA=D**DB	D=0, DB ≤0	If DB=0, DA=1 If DB<0, DA=•	A, B, D, DB
246	CA=C**I	C=0 + 0i, I≤0	If I=0, C=1 + 0i IF I < 0, C=• + 0i	A, B, C, I
247	CDA=CD I	C=0 + 0i, I≤0	If I=0, C=1 + 0i If I < 0, C=•+ 0i	A, B, CD, I
248 [5]	Q=QA**J	QA=0, J≤0	J < 0, Q=• J=0, Q=1	A, B, QA, J
249	Q=QA**QB	QA=0, QB≤0	QB < 0, Q=• QB=0, Q=1	A, B, QA, QB
		QA < 0, QB≠0	Q=$\lvert QA\rvert$**QB	
250	Q=QA**QB	$\log_2$(QA) × QB ≥ 252	Q=•	A, B, QA, QB
251	Y=SQRT (X)	X < 0	Y=$\lvert X\rvert^{1/2}$	A, B, X
252	Y=EXP (X)	X > 174.673	Y•	A, B, X

253	Y=ALOG (X)	X=0
		X < 0
			Y=-•
			Y=log$\lvert X\rvert$
				A, B, X
				A, B, X

	Y=ALOG10 (X)	X=0
		X lt 0
			Y=-•
			Y=$\log_{10}\lvert X\rvert$
				A, B, X

254	Y=COS (X)
	Y=SIN (X)
		$\lvert X\rvert \geq (2^{18})\pi$	Y = $\sqrt{2/2}$	

255	Y=ATAN2 (X,XA)	X=0, XA=0	Y=0	A, B, X, XA

256	Y=SINH (X)
	Y=COSH (X)
		$\lvert H\rvert \geq 175.366$	Y=(SIGN of X)
			Y=•
				A, B, X

257	Y=ASIN (X)	$\lvert X\rvert > 1$	If X > 1.0, ASIN (X)=π/2 If X < - 1.0, ASIN (X)= − π/2	
	Y=ACOS (X)		If X > 1.0, ACCOS=0 If X <- 1.0, ACOS=π	

258	Y=TAN(X)
	Y=COTAN(X)
		$\lvert X\rvert \geq (2^{18})\pi$	Y=1	

	Y=COTAN (X)	X=0	Y=.	
260	Q=2 QA	QA >252	Q=•	A, B, QA
261	DA=DSQRT (D)	D < 0	DA=$\lvert D\rvert^{1/2}$	A, B, D
262	DA + DEXP (D)	D > 174.673	D=•	A, B, D

263	DA=DLOG (D)	D=0
		D < 0
			DA=0•
			DA=log$\lvert X\rvert$	

	DA=DLOG10 (D)	D=0
		D < 0
			DA=-•
			DA=$\log_{10}\lvert X\rvert$
				A, B, D

Figure 55 (Part 1 of 3). Corrective Action after Mathematical Subroutine Error

Error Code	FORTRAN Reference[1]	Invalid Argument Range	Options Standard Corrective Action[2], [3]	Options Parameters Passed to User Exit[4]

264	DA=DSIN (D)
	DA=DCOS (D)
		$\|D\| \geq (2^{50})\pi$	$DA=\sqrt{2/2}$	A, B, D

265	DA=DATAN2 (D,DB)	D=0, DB=0	DA=0	A, B, D, DB

266	DA=DSINH (D)
	DA=DCOSH (D)
		¡D| ≥ 175.366	DA=(SIGN of X)•
			DA=•
				A, B, D

267	DA=DASIN (D)	|D| > 1	If D > 1.0, DASIN=π/2 If D < -1.0, DASIN= − π/2	
	DA=DACOS (D)		If D > 1.0, DACOS (D)=0 If D < - 1.0, DACOS (D)=π	

268	DA=DTAN (D)
	DA=DCOTAN (D)
		$\|X\| \geq (2^{50})\pi$	DA=1	A, B, D

	DA=DCOTAN (D)	D=0	DA=•	A, B, D
270[6]	CQ=CQA**J	CQA=0 + Oi J ≤ 0	J=0, CQ=1 + 0,i J < 0, CQ=•+ 0,i	A, B, CQA, J
271[7]	Z=CEXP (C)	$X_1 < 174.673$	Z=•(cos X_2 + iSIN X_2)	A, B, C
272	Z=CEXP (C)	$\|X_2\| \geq (2^{18})\pi$	$Z=e^x$ + 0i	A, B, C
273	Z=CLOG (C)	C=0 + 0i	Z=-• + 0i	A, B, C
274	Z=CSIN (C)	$\|X_1\| \geq (2^{18})\pi$	Z=0 + SINH $(X_2)\pi$	A, B, C
	Z=CCOS (C)		Z=COSH (X_2) + 0i	A, B, C
275	Z=CSIN (C)	$X_2 < 174.673$	$Z=\frac{\bullet}{2}$ (SIN X_1 + iCOS X_1)	A, B, C
	Z=CCOS (C)		$Z=\frac{\bullet}{2}$ (COS X_1 − iSIN X_1)	A, B, C
275	Z=CSIN (C)	$X_2 < -174.673$	Z= $\frac{\bullet}{2}$ (SIN X_1 − iCOS X_1)	A, B, C
	Z=CCOS (C)		Z= $\frac{\bullet}{2}$ (COS X_1 + iSIN X_1)	A, B, C
276[8]	Z=CQEXP (CQ)	$X_1 > 174.673$	Z=•(COS X_2 − iSIN X_2)	A, B, CQ
277	Z=CQEXP (CQ)	$\|X_2\| > 2_{100}$	$Z=e_{x1}$ − 0i	A, B, CQ
278	Z=CQLOG (CQ)	CQ=0 + 0i	Z=-• + 0i	A, B, CQ

279	Z=CQCOS (CQ)
	Z=CQCOS (CQ)
		$\|X_1\| \geq 2^{100}$	Z=0 + DSINH (X_2)i
			Z=DCOSH (X_2) + 0i
				A, B, CQ

280	Z=CQSIN (CQ)	$X_2 > 174.673$	$Z=\frac{\bullet}{2}$ (SIN X_1 + iCOS X_1)	A, B, CQ
	Z=CQCOS (CQ)		$Z=\frac{\bullet}{2}$ (COS X_1 = iSIN X_1)	A, B, CQ

	Z=CQSIN (CQ)
	Z=CQCOS (CQ)
		$X_2 < -174.673$	$Z=\frac{\bullet}{2}$ (COS X_1 = iSIN X_1)
			$Z=\frac{\bullet}{2}$ (COS X_1 = iSIN X_1)
				A, B, C Q

281[9]	Z=CDEXP (CD)	$X_1 > 174.673$	Z=•(COS X_2 + iSIN X_2)	A, B, CD
282	Z=CDEXP (CD)	$\|X_2\| \geq (2^{50})\pi$	$Z=e^{x}{}_1$ + 0i	A, B, CD
283	Z=CDLOG (CD)	CD = 0 + 0i	Z=- • + 0i	A, B, CD
284	Z=CDSIN (DC)	$\|X_1\| \geq (2^{50})\pi$	Z=0 + SINH (X_2)i	A, B, CD

Figure 55 (Part 2 of 3). Corrective Action after Mathematical Subroutine Error

Error Code	FORTRAN Reference[1]	Invalid Argument Range	Options Standard Corrective Action[2], [3]	Options Parameters Passed to User Exit[4]
	Z=CDCOS (CD)		Z=COSH (X_2) + 0i	A, B, CD
285	Z=CDSIN (CD)	$X_2 > 174.673$	$Z=\frac{\bullet}{2}(\text{SIN } X_1 + i\text{COS } X_1)$	A, B, CD
	Z=CDCOS (CD)		$Z=\frac{\bullet}{2}(\text{COS } X_1 - i\text{SIN } X_1)$	A, B, CD
	Z=CDSIN (CD)	$X_2 < - 174.673$	$Z=\frac{\bullet}{2}(\text{SIN } X_1 - i\text{COS } X_1)$	A, B, CD
	Z=CDCOS (CD)		$Z=\frac{\bullet}{2}(\text{COS } X_1 + i\text{SIN } X_1)$	A, B, CD
289	QA=QSQRT (Q)	Q < 0	QA=$\lvert Q\rvert^{1/2}$	A, B, Q
290	Y=GAMMA (X)	X≤ 2^{-252} or X≥ 57.5744	Y=•	A, B, X
291	Y=ALGAMA (X)	X≤ 0 or X ≥ 4.2937 × 10^{73}	Y=•	A, B, X
292	QA=QEXP (Q)	Q > 174.673	QA=•	A, B, Q

293	QA=QLOG (Q)	Q=0
		Q < 0
			QA=-•
			QA=log$\lvert X\rvert$
				A, B, Q

	QA=QLOG10 (Q)	Q=0
		Q < 0
			QA=•
			QA=$\log_{10}\lvert X\rvert$
				A, B, Q
				A, B, Q

294	QA=QSIN (Q)
	QA=QCOS (Q)
		$\lvert Q\rvert \geq 2^{100}$	QA=$\sqrt{2/2}$	A, B, Q

295	QA=QATAN2 (Q, QB)	Q=0, QB=0	QA=0	A, B, Q, QB

296	QA=QSINH (Q)
	QA=QCOSH (Q)
		$\lvert Q\rvert \geq 175.366$	QA=•(SIGN Q)
			QA=•
				A, B, Q

297	QA=QARSIN (Q)	$\lvert Q\rvert > 1$	If Q > 1.0, QARSIN=π/2 If Q < -1.0, QARSIN=π/2	A, B, Q A, B, Q
	QA=QARCOS (Q)		If Q > 1.0, QARCOS (Q)=0 If Q < -1.0, QARCOS (Q)=π	

298	QA=QTAN (Q)
	QA=QCOTAN (Q)
		$\lvert Q\rvert > 2^{100}$	QA=1	A, B, Q

299	QA=QTAN (Q)	Q is too close to an odd multiple of π/2	QA=•	A, B, Q
	QA=QCOTAN (Q)	Q is too close to a multiple of π	QA=•	A, B, Q
300	DA=DGAMMA (D)	D≤ 2^{-252} or D≥ 57.5774	DA=•	A, B, D
301	DA=DLGAMA (D)	D≤ 0 or D≥ 4.2937 10^{73}	DA=•	

Figure 55 (Part 3 of 3). Corrective Action after Mathematical Subroutine Error

Notes to Figure 55:

[1] The variable types are as follows:

Variable	Type
A,B	INTEGER*4
I,J,K	INTEGER*4
X,XA,Y	REAL*4
D,DA,DB	REAL*8
C,CA	COMPLEX*8
Q,QA,QB	REAL*16
CQ,CQA	COMPLEX*32
Z,X_1,X_2	Complex variables to be given the length of the function argument when they appear.
CD,CDA	COMPLEX*16

[2] The largest number that can be represented in floating point is indicated by the symbol •.

[3] The value e= 2.7183 (approximately).

[4] The user-supplied answer is obtained by computation of the function using the value set by the user routine for the parameters listed.

[5] The values of the base and exponent are limited to values where BASE**exponent and BASE**(-exponent) are representable.

[6] For error 270, CQA $= X_1 + iX_2$

[7] For errors 271 through 275, C $= X_1 + iX_2$

[8] For errors 276 through 280, CQ $= X_1 + iX_2$

[9] For errors 281 through 285, CD $= X_1 + iX_2$

Chapter 10. Multitasking Facility (MTF) Subroutines

The MTF subroutines are supplied as part of the VS FORTRAN Version 2 Library. When accessed in a main task program by the appropriate entry name in a CALL statement, they perform the multitasking functions. The multitasking capability provided by MTF is available only when running under the MVS or MVS/Extended Architecture (MVS/XA) operating systems.

For more information on coding parallel subroutines for multitasking, see *VS FORTRAN Version 2 Programming Guide*.

NTASKS Subroutine

The NTASKS subroutine returns the number of subtasks specified with the AUTOTASK keyword in the PARM parameter of the EXEC statement for the job step.

Syntax

CALL NTASKS(*n*)

n

is an integer variable or an integer array element of length 4 in the program unit.

The values returned in *n* have the following meanings:

Value	Meaning
0	The AUTOTASK keyword was not specified.
1 - 99	The number of subtasks specified with the AUTOTASK keyword.

Notes:

1. NTASKS may be called in a main task program as often as desired. However, it will always return the same value, because the number of subtasks does not change during the execution of a program.
2. NTASKS may be called only from a main task program. If it is called from a parallel subroutine, the program will be terminated with a return code of 16.
3. If NTASKS is called by a program running under CMS, *n* will be set to 0.

DSPTCH Subroutine

The DSPTCH subroutine schedules a parallel subroutine for execution in a subtask. You may call DSPTCH as many times as necessary. Eventually, you must call SYNCRO to wait for all the parallel subroutines to finish executing.

Syntax

CALL DSPTCH (*subrname* [, *arg*$_1$ [, *arg*$_2$]...])

subrname

specifies the 1- to 8-character name of the parallel subroutine to be scheduled.

If the subroutine is a FORTRAN subroutine, and the name is longer than 7 characters, *subrname* must be the shortened form of the subroutine name. For a description of the shortened form of long global names, see the note on page 8.

For FORTRAN Language Level 77, *subrname* is a character expression.

For Language Level 77, the character expression *subrname* may be longer than the subroutine name it will contain. If it is, the name must be left-adjusted within the field and padded on the right with blanks to at least 8 characters. The value, after trailing blanks are removed, must be 8 characters in length.

For FORTRAN Language Level 66, *subrname* must be specified as a character constant of 8 or more characters. The name must be left-adjusted within the field and padded on the right with blanks to a length of 8 characters. The value, after trailing blanks are removed, must be 8 characters in length.

[,*arg*$_1$[,*arg*$_2$]...]

specifies the actual arguments that are being supplied to the parallel subroutine.

Each argument may be:

- A variable
- An array element
- An array name
- A constant

The following must **not** be used for the actual arguments being supplied to the parallel subroutine subprogram:

- Expressions requiring evaluation; for example, A+2*B**3
- Function names
- Subroutine names
- Alternate return specifiers; that is, the form *n, where *n* is a statement label.

Notes:

1. You may call DSPTCH from your main task program as often as necessary to schedule parallel subroutines for execution. However, if, prior to calling SYNCRO you call DSPTCH more times than there are subtasks available, each call in excess of the number of subtasks will cause your main task program to wait until one of the previously-scheduled parallel subroutines has finished executing.
2. DSPTCH usually returns to the main task program before the scheduled parallel subroutine has completed execution. Therefore, you must call SYNCRO to ensure that your parallel subroutines have completed execution.
3. DSPTCH may be called only from a main task program. If you call it from a parallel subroutine, the program will be terminated with a return code of 16.
4. If DSPTCH is called by a program executing under MVS or MVS/XA and the AUTOTASK keyword was not specified, the program will be terminated with a return code of 16.
5. If DSPTCH is called by a program running under CMS, the program will be terminated with a return code of 16.

SYNCRO Subroutine

The SYNCRO subroutine causes the main task program to wait until all scheduled parallel subroutines have completed execution.

Syntax

```
CALL SYNCRO
```

SYNCRO has no arguments.

Notes:

1. You may call SYNCRO in a main task program as often as necessary. If there are no parallel subroutines scheduled when SYNCRO is called, the call is ignored.
2. SYNCRO may be called only from a main task program. If you call it from a parallel subroutine, the program will be terminated with a return code of 16.
3. If SYNCRO is called by a program executing under MVS or MVS/XA and the AUTOTASK keyword was not specified, the call is ignored.
4. If SYNCRO is called by a program running under CMS, the call is ignored.

SHRCOM Subroutine

The SHRCOM subroutine allows you to designate a dynamic common block as shareable among the main task program and the parallel subroutines.

Syntax

CALL SHRCOM (*dyncom*)

dyncom

is a character expression whose value, when trailing blanks are removed is the name of a dynamic common block. The dynamic common, *dyncom*, must be defined within some program unit in the main task program that has been entered at least once before SHRCOM is called. If *dyncom* has not been defined, an error will be detected.

Notes:

1. If a parallel subroutine is to use a shared copy of a dynamic common block, the main task program must designate that common block as shareable before the subroutine is scheduled.
2. If a program unit within a parallel subroutine refers to a dynamic common block that has not been designated as shareable, a dynamic common block will be acquired for the exclusive use of that subtask, which will be available to all program units within that subtask.
3. If SHRCOM is called by the main task program to make shareable a dynamic common block that has already been acquired for the exclusive use of a subtask, an error will be detected.
4. A dynamic common block that is shared among the main task program and the parallel subroutines may be the virtual storage window that corresponds to part of a data-in-virtual object. However, all of the DIV service calls are restricted to the main task program.
5. Static common blocks cannot be shared among the main task program and parallel subroutines.

Appendix A. Source Language Flaggers

ANS Language (FIPS) Flagger

The VS FORTRAN Version 2 compiler can flag FORTRAN statements that do not conform to the syntax of the Full or Subset ANS FORTRAN Standard (FORTRAN 77). The FIPS compiler option specifies whether this flagging is to be performed.

Systems Application Architecture Flagger

The VS FORTRAN Version 2 compiler can also flag FORTRAN statements that are not a part of the Systems Application Architecture (SAA) Common Programming Interface (CPI) FORTRAN language definition. The SAA compiler option specifies whether this flagging is to be performed.

For more information about the compiler options, see the *VS FORTRAN Version 2 Programming Guide.*

Items Flagged for FIPS and SAA

The table in Figure 56 lists the major elements of the VS FORTRAN Version 2 language. The table shows whether:

- An element is flagged for not conforming to the FORTRAN 77 standard (FIPS)
- An element is flagged for not conforming to the Systems Application Architecture FORTRAN definition (SAA)

Statement or Feature	Use	Flagged by FIPS?	Flagged by SAA?
Ampersand (&)	As special character	Yes	Yes
ASSIGN	Assigns GOTO targets	No	No
	Assigns FORMAT labels	No	No
Assignment statements	Assign values to arithmetic and logical data items	No	No
	Assign values to character data items	No	No
Asynchronous I/O	Read/write in asynchronous mode	Yes	Yes
AT	Specifies beginning of debugging packet	Yes	Yes
BACKSPACE	Repositions file at previous record	No	No
BLOCK DATA	Identifies a data subprogram	No	No
CALL	Transfers control to a subroutine	No	No
Character data type	For character (string) data	No	No

Figure 56 (Part 1 of 4) Major Elements of the VS FORTRAN Version 2 Language

Statement or Feature	Use	Flagged by FIPS?	Flagged by SAA?
CLOSE	Disconnects file from a program	No	No
Columns 1 to 5	Non-blank on continuation line	Yes	Yes
COMMON	Defines storage shared between programs	No	No
	Both character and noncharacter data in one block	No	No
Complex data type	Complex numbers of single precision	No	No
	Complex numbers of double and extended precision	Yes	No
CONTINUE	Non-operational executable statement for programming convenience	No	No
Currency symbol ($)	In names	Yes	Yes
DATA	Initializes variables and array elements	No	No
	Initializes variables and arrays with implied DO loops if desired	No	No
DEBUG and END DEBUG	Delimit the debugging packet portion of a program	Yes	Yes
DEFINE FILE	Specifies a direct-access file	Yes	Yes
DELETE	Deletes record from a KSDS file	Yes	Yes
DIMENSION	Defines arrays of up to three dimensions	No	No
	Defines arrays of up to seven dimensions	No	No
	Defines arrays with adjustable size	No	No
	Defines arrays with explicit lower bounds (which can be positive or negative)	Yes	No
Direct-access I/O	Read/write by record number	No	No
DISPLAY	Displays data within a debugging packet	Yes	Yes
DO	Gives a convenient way to program loops (using integer DO variables)	No	No
	Real and double precision DO variables are allowed; negative increment parameter is allowed	No	No
DO WHILE	Initiates processing of program loops based on evaluation of a logical expression	Yes	Yes
EJECT	Starts new page of source listing	Yes	Yes
END	Marks end of program unit	No	No
	Terminates program processing	No	No
END DO	Terminates processing of a DO or DO WHILE loop	Yes	Yes
end of line commentary	The "!" indicates the beginning of a comment	Yes	Yes
ENDFILE	Writes end-of-file record	No	No
ENTRY	Specifies alternate entry points into subprograms	No	No
EQUIVALENCE	Defines shared storage	No	No
	Relates character and noncharacter data	No	No
Explicit type statements	Define data types of specific variables	No	No
Expressions	Manipulate arithmetic, relational, or logical items, or other expressions	No	No
	Manipulate character items or arithmetic double precision or complex items	No	No

Figure 56 (Part 2 of 4). Major Elements of the VS FORTRAN Version 2 Language

Statement or Feature	Use	Flagged by FIPS?	Flagged by SAA?
EXTERNAL	Defines linked subprograms	No	No
FIND	Locates next input record	Yes	Yes
FORMAT	Defines record formats	No	No
	Character constants and run-time formats allowed	No	No
Free-form source	Relaxes format rules for source program	Yes	Yes
FUNCTION	Identifies a function subprogram	No	No
GENERIC	Automatic function selection	Yes	Yes
GO TO	Specifies transfers of control	No	No
Hexadecimal constants	For initializing data values	Yes	Yes
Hollerith constants	For initializing integer variables	Yes	Yes
	As arguments in CALL statements	Yes	Yes
	As character strings in FORMAT statements	Yes	No
IF	Specifies alternate paths of processing, using arithmetic and logical IF versions	No	No
	Block IF version, using ELSE, ELSE IF, and END IF	No	No
IMPLICIT	Types groups of variables	No	No
INCLUDE	Copies pre-written source statements into program	Yes	No
Integer data type	For integer numbers	No	No
INQUIRE	Retrieves information about a file	No	No
Internal files	Easy data conversion	No	No
Intrinsic functions	Supply arithmetic and generic functions	No	No
	Supply character and bit functions	No	No
INTRINSIC	Explicitly defines intrinsic functions	No	No
I/O status indicator	Determine success of input/output statement	No	No
Keyed I/O	Read/write by record key value	Yes	Yes
Length fields	Optional specification for data types	Yes	No
List-directed I/O	Read/write formatted data without FORMAT statement	No	No
'Literal constants'	Literal constants enclosed in apostrophes	No	No
Logical data type	True/false values	No	No
Mixed-mode expressions	For mixing of data types	No	No
NAMELIST	Read/write referencing named list	Yes	Yes
OPEN	Connects files to a program; error routines can be specified	No	No
PARAMETER	Establishes names for constants	No	No
PAUSE	Suspends program processing temporarily	No	No
PRINT	Installation-dependent write statement	No	No
PROGRAM	Names a main program	No	No
PUNCH	Installation-dependent write statement	Yes	Yes
Quotation mark	Double quote (") as special character	Yes	Yes
READ	Reads a record from a file	No	No

Figure 56 (Part 3 of 4). Major Elements of the VS FORTRAN Version 2 Language

Statement or Feature	Use	Flagged by FIPS?	Flagged by SAA?
Real data type	Single precision floating-point numbers	No	No
	Double precision floating-point numbers	No	No
	Extended precision floating-point numbers	Yes	Yes
Real subscripts	Expressions with floating-point numbers can be used as subscripts	Yes	Yes
RETURN	Returns control to a calling program	No	No
REWIND	Repositions to beginning of file	No	No
REWRITE	Rewrites record in a KSDS file	Yes	Yes
SAVE	Saves values after a called program completes executing	No	No
Sequential I/O	Read/write sequential files	No	No
Statement functions	Convenient programming of expressions	No	No
STOP	Terminates program processing	No	No
SUBROUTINE	Identifies a subroutine subprogram	No	No
Symbolic names	31 characters long	Yes	No
TRACE ON/OFF	Traces specific portions of a program	Yes	Yes
Underscore character (_)	In names	Yes	No
VSAM I/O	Supports ESDS, RRDS, and KSDS files	Yes	Yes
WRITE	Writes a record into a file	No	No

Figure 56 (Part 4 of 4). Major Elements of the VS FORTRAN Version 2 Language

Appendix B. Assembler Language Information

The mathematical and service routines (including the vector intrinsic elementary functions) in the VS FORTRAN Version 2 library can be used by the assembler language programmer. To be successful, you need to do three things:

- Make the library available to the linkage editor or loader.
- Set up proper calling sequences.
- Supply correct parameters.

Library Availability

The assembler language programmer must arrange for the desired routines (modules) to be taken from the VS FORTRAN Version 2 library and brought into main storage, usually as a part of the programmer's load module. This can be done by employing the techniques described in the appropriate publications for your operating system.

For example, in MVS, the VS FORTRAN Version 2 library could be made part of the automatic call library for the linkage editor by using the following job control statements. Note that, to assemble vector mnemonics, you must use Assembler H Version 2 Release 1.0 with the appropriate Program Temporary Fixes (PTFs).

In Link Mode:

```
//jobname JOB ...operands
//ASM EXEC ASMHCLG,PARM.L='LET,LIST,MAP'
//C.SYSIN DD *

  (assembler language program source deck)

/*
//L.SYSLIB DD DSNAME=SYS1.VSF2LINK,DISP=SHR
//          DD DSNAME=SYS1.VSF2FORT,DISP=SHR
/*
```

In Load Mode:

```
//jobname JOB ...operands
//ASM EXEC ASMHCLG,PARM.L='LET,LIST,MAP'
//C.SYSIN DD *

  (assembler language program source deck)

/*
//L.SYSLIB DD DSNAME=SYS1.VSF2FORT,DISP=SHR
//G.STEPLIB DD DSNAME=SYS1.VSF2FORT,DISP=SHR
/*
```

Library routines requested in the source program would then be made available to the linkage editor for inclusion in the load module. This is made possible by using the name of the library as the data set name in the SYSLIB DD statement.

Initializing the Execution Environment

If your main program is not written in FORTRAN and it calls VS FORTRAN Version 2 library routines or other FORTRAN routines, the calling program must first initialize the execution environment. For more information on how to do this, see *VS FORTRAN Version 2 Programming Guide.*

Calling Sequences

Two general methods of calling are possible:

- Code an appropriate macro instruction, such as CALL.
- Code assembler language branch instructions.

In all cases, a save area must be provided that:

- Is aligned on a fullword boundary
- Is 18 words (72 bytes) in length
- Has its address in general register 13 at the time of the CALL macro instruction or branch

Figure 57 on page 348 through Figure 61 on page 350 contain assembler information for VS FORTRAN Version 2 routines:

Assembler Information	Figure
Explicitly called mathematical routines	Figure 57
Implicitly called mathematical routines	Figure 58
Implicitly called character routines	Figure 59
Service routines	Figure 60
Explicitly called bit functions	Figure 61

Figure 62 on page 351 is an example of Assembler language calling sequence with CALL macro. Figure 63 on page 351 is an example of Assembler language calling sequence with BALR.

For vector intrinsic elementary functions, the result registers are listed in Figure 64 and Figure 65 on page 352, and Figure 66 on page 352.

Figure 67 on page 354 contains assembler information for vector intrinsic elementary functions.

Figure 68 on page 355 and Figure 69 on page 356 show calling sequences for a specific example: how to find the square root of a vector of 10 arguments using vector registers. Figure 68 shows the sequence with vector mask mode off, and Figure 69 with vector mask mode on. The library square root routine (entry name V#SQRT or W#SQRT) is invoked, using assembler language statements.

Notes:

1. For performance reasons, VS FORTRAN Version 2 routines use certain floating-point registers (see Figure 58 on page 349), but do not save and restore original register contents. If you want floating-point register contents retained, you must save them before calling the routine and restore it on return.

2. Performance Hint: To speed up save area loads and stores, align the save area within your routine on a doubleword boundary, as shown below:

   ```
               CNOP 4,8
   SAV         DC   18F'0'
   ```

   Execution of the `STM 14,12,12(13)` instruction in a called routine will be faster using this alignment.

Assembler Language Calling Sequence

When a branch instruction, rather than a CALL macro instruction, is used to invoke a routine, several additional conventions must be observed:

- An argument (parameter) address list must be assembled on a fullword boundary. It consists of one 4-byte address constant for each argument, with the last address constant containing a 1 in its high order bit.
- The address of the first item in this argument address list must be in general register 1.
- The address of the entry point of the called routine must be in general register 15.
- The address of the point of return to the calling program must be in general register 14.
- The address of the save area (72 bytes) must be in general register 13.

Supplying Correct Parameters

Arguments must be of the proper type, length, and quantity, and, in certain cases, within a specified range, for the routine called.

For mathematical and character routines, this information can be found in Figure 26 on page 246 through Figure 32 on page 257.

- INTEGER*4 denotes a signed binary number 4 bytes long.
- REAL*4, REAL*8, and REAL*16 are normalized floating-point numbers 4, 8, and 16 bytes long, respectively.
- COMPLEX*8, COMPLEX*16, and COMPLEX*32 are complex numbers 8, 16, and 32 bytes long, respectively, represented by a pair of floating-point numbers whose first half contains the real part, and whose second half contains the imaginary part. Each part is a normalized floating-point number.
- Four-byte argument types must be aligned on fullword boundaries; 8-byte, 16-byte, and 32-byte types must be aligned on doubleword boundaries.

Argument information for nonmathematical routines can be found under Chapter 7, "Service Subroutines" on page 269.

Error messages resulting from incorrect arguments are explained in Appendix D, "Library Procedures and Messages" on page 375.

Mathematical Routine Results

Each mathematical routine returns a single answer of a type listed in Figure 26 on page 246 through Figure 32 on page 257.

- Integer answers are returned in general register 0.
- Real answers are returned in floating-point register 0.
- Complex answers are returned in floating-point registers 0 and 2.

Result registers are listed by routine entry name in Figure 57 on page 348 and Figure 58 on page 349.

For vector intrinsic elementary functions, the result registers are listed in Figures 64, 65, and 66.

For extended-precision mathematical routines, results are always returned in the floating-point registers:

- 0 and 2 for REAL*16 results
- 0 and 2, 4 and 6 for COMPLEX*32 results

The location and form of the service subroutine results can be determined from the discussion under Chapter 7, "Service Subroutines" on page 269.

Space Considerations

Many of the mathematical routines require other mathematical routines for some of their calculations. In addition, most of the routines use the input/output, error processing, and interruption library subroutines. In case of errors, the vector mathematical routines use the scalar routines to locate the incorrect element of the vector.

Floating Point Registers Used by Routines

Routine Entry Name	Result Registers	Intermediate Registers
ABS,DABS,IABS,QABS	0	2
ACOS,ASIN	0	2,4,6
ACOS,ASIN[1]	0	2,4
AIMAG,DIMAG,QIMAG	0	2,4,6
AINT,DINT,QINT	0	2,4,6
ALGAMA,GAMMA,LGAMMA	0	2,4,6
ALOG,LOG	0	2
ALOG,LOG[1]	0	2,4,6
ALOG10,LOG10	0	2
ALOG10,LOG10[1]	0	2,4,6
AMOD,DMOD,QMOD	0	2,4,6
ANINT,DNINT,NINT,IDNINT	0	2,4,6
ATAN	0	2,4,6
ATAN2	0	2,4
ATAN,ATAN2[1]	0	2,4,6
CABS	0	2,4,6
CABS[1]	0,2	6
CCOS,CSIN	0,2	4
CDABS	0	2,4,6
CDABS[1]	0,2	4,6
CDCOS,CDSIN	0,2	4,6
CDEXP	0,2	4,6
CDLOG	0,2	4,6
CDSQRT	0,2	4,6
CEXP	0,2	4,6
CLOG	0,2	4,6
CONJG,DCONJG,QCONJG	0	2,4,6
COS	0	2,4
COS[1]	0	2,4
COSH,SINH	0	2,4
COTAN,TAN	0	2,4,6
COTAN,TAN[1]	0	2,4
CQABS	0,2,4,6	
CQCOS,CQSIN	0,2,4,6	
CQEXP	0,2,4,6	
CQLOG	0,2,4,6	
CQSQRT	0,2,4,6	
CSQRT	0,2	4,6
DACOS,DASIN	0	2,4,6
DACOS,DASIN[1]	0	2,4
DATAN,DATAN2	0	2,4,6
DATAN,DATAN2[1]	0	2,4,6
DCOS	0	2,4,6
DCOS[1]	0	2,4
DCOSH,DSINH	0	2,4,6
DCOTAN,DTAN	0	2,4,6
DCOTAN,DTAN[1]	0	2,4,6
DDIM,DIM,IDIM,QDIM	0	2,4,6
DERF,DERFC	0	2,4,6
DEXP	0	2,4
DEXP[1]	0	2
DGAMMA,DLGAMA	0	2,4,6
DLOG	0	2,4,6
DLOG[1]	0	2,4,6
DLOG10	0	2,4,6
DLOG10[1]	0	2,4,6
DPROD	0	2
DSIGN,ISIGN,SIGN,QSIGN	0	2,4,6

Figure 57 (Part 1 of 2). Explicitly Called Mathematical Routine Assembler Information

Routine Entry Name	Result Registers	Intermediate Registers
DSIN	0	2,4,6
DSIN[1]	0	2,4
DSQRT	0	2,4,6
DSQRT[1]	0	2,4
DTANH	0	2,4,6
ERF,ERFC	0	2,4,6
EXP	0	2
EXP[1]	0	
MOD	0[2]	
QARCOS,QARSIN	0,2	4,6
QATAN,QATAN2	0,2	4,6
QCOS,QSIN	0,2	4,6
QCOSH,QSINH	0,2	4,6
QCOTAN,QTAN	0,2	4,6
QERF,QERFC	0,2	4,6
QEXP,QLOG,QLOG10	0,2	4,6
QSQRT	0,2	4,6
QTANH	0,2	4,6
SIN	0	2,4
SIN[1]	0	2,4
SQRT	0	2,4
SQRT[1]	0	2
TANH	0	2,4,6

Figure 57 (Part 2 of 2). Explicitly Called Mathematical Routine Assembler Information

Notes to Figure 57:

[1] Alternative mathematical library subroutines

[2] General register

Routine Entry Name	Result Registers[1]	Intermediate Registers[1]
CDDVD#	0,2	4,6
CDMPY#	0,2	4,6
CDVD#	0,2	4,6
CMPY#	0,2	4,6
CQDVD#,CQMPY#	0,2,4,6	
CXMPR#	0[2]	
FCDCD#	0,2,4,6	
FCDXI#	0,2	
FCQCQ#	0,2,4,6	
FCQXI#	0,2,4,6	
FCXPC#	0,2,4,6	
FCXPI#	0,2	
FDXPD#	0	2,4,6
FDXPD#[1]	0	
FDXPI#	0	
FIXPI#	0[2]	
FQXPI#	0,2	4,6
FQXPQ#,FQXP2#	0,2	4,6
FRXPI#	0	
FRXPR#	0	2,4,6
FRXPR#[1]	0	

Figure 58. Implicitly Called Mathematical Routine Assembler Information

Notes to Figure 58:

[1] Alternative mathematical library subroutines

[2] General register

Routine Entry Name(s)
CCMPR#
CHAR, ICHAR, LEN
CMOVE#
CNCAT#
INDEX
LGE, LGT, LLE, LLT

Figure 59. Implicitly Called Character Routine Assembler Information

Notes follow Figure 61.

Routine Entry Name(s)
ASSIGNM
CDUMP, CPDUMP
CLOCK, CLOCKX
DATIM, DATIMX
DSPAN#, DSPN2#, DSPN4#
DUMP, PDUMP
DVCHK
DYCMN#
EXIT
FILEINF
OVERFL
SDUMP
SYSRCS, SYSRCT, SYSRCX, SYSABD, SYSABN
XUFLOW

Figure 60. Service Routine Assembler Information

Routine Entry Name(s)
IBCLR, IBSET, BTEST, ISHFT
IOR, IEOR, NOT, IAND

Figure 61. Explicitly Called Bit Function Assembler Information

Notes to Figures 59, 60, and 61:

No floating-point registers are used in:

- Implicitly called character routines
- Service routines
- Explicitly called bit functions

```
*        The following instructions show the use of a CALL
*        macro to call the library square root routine.
         .
         .
         .
         CALL   SQRT,(AMNT),VL          (See Notes 1, 2)
         STE    0,ANSWER
         .
         .
         .
AMNT     DC     E'144'
ANSWER   DC     E'0'
         .
         .
         .
```

Figure 62. Example of Assembler Language Calling Sequence with CALL Macro

Notes to Figure 62:

Notes:

1. The VL operand in CALL indicates that the macro expansion should flag the end of the parameter list.
2. If you expect to execute your program on an MVS/XA system, you must assemble your program to make the MVS/XA version of the CALL macro available.

```
*        The following instructions show the use of a BALR
*        sequence to call the library square root routine.
         .
         .
         .
         LA     1,ARG
         L      15,ENTRY
         BALR   14,15
         STE    0,ANSWER2
         .
         .
         .
ENTRY    DC     V(SQRT)
ANSWER2  DC     E'0'
         .
         .
         .
ARG      DC     A(AMNX+X'80000000')
         .
         .
         .
AMNX     DC     E'144'
```

Figure 63. Example of Assembler Language Calling Sequence with BALR

Calling Vector Intrinsic Elementary Functions

The vector intrinsic elementary functions can be called using either the CALL macro, or a BALR sequence. Both methods are shown in Figure 68 on page 355 and Figure 69 on page 356. No parameter list is passed to these routines. Instead, all data is passed through vector hardware registers.

Required Contents of Registers

The following sections describe the required contents of these registers:

- Vector registers
- Vector count register
- Vector mask register
- Vector interruption index

Vector Registers: The vector registers in which arguments are passed and in which results are returned by the vector intrinsic elementary functions depend on the number and data types of the arguments. Figures 64, 65, and 66 indicate which vector registers are used.

Function	Vector Registers	
Type	Real Part	Imaginary Part
REAL*4	0	
REAL*8	0-1	
COMPLEX*8	0	2
COMPLEX*16	0-1	2-3
INTEGER*4	0	
LOGICAL*4	0	

Figure 64. Vector Intrinsic Elementary Function Result Vector Registers

Argument Type	Real Part	Imaginary Part
REAL*4	14	
REAL*8	14-15	
COMPLEX*8	12	14
COMPLEX*16	12-13	14-15

Figure 65. Argument Vector Registers for Functions of One Argument

Arg. 1 Type	Real Part	Imaginary Part	Arg. 2 Type	Real Part	Imaginary Part
REAL*4	12		REAL*4	14	
REAL*4	12		INTEGER*4	14	
REAL*8	12-13		REAL*8	14-15	
REAL*8	12-13		INTEGER*4	14	
COMPLEX*8	8	10	COMPLEX*8	12	14
COMPLEX*8	10	12	INTEGER*4	14	
COMPLEX*16	8-9	10-11	COMPLEX*16	12-13	14-15
COMPLEX*16	10-11	12-13	INTEGER*4	14	
INTEGER*4	12		INTEGER*4	14	

Figure 66. Argument Vector Registers for Functions of Two Arguments

The vector registers containing the arguments and results are not preserved across calls to these functions. Furthermore, the odd half of the vector register pairs used for short precision, integer, or logical arguments or results are not preserved.

Some vector registers are used for intermediate calculations. This information is indicated in the figures that follow. The vector registers used for intermediate calculations are preserved across the call if the caller passes a byte in the rightmost part of general register 0, indicating which vector registers are to be preserved across the call. The i-th bit in the byte should be set to 1 if the following vector register pair is to be preserved:

```
2i - 2i+1, i=0, ..., 7
```

For example, if vector register pairs 2 and 3 and 6 and 7 must be saved across a call to VD#TAN, byte 01010000 should be passed. Registers containing the arguments and results are not preserved, regardless of the byte passed in general register 0.

Note that, if X'FF' is passed, all but the argument and result vector registers are preserved. The byte X'00' should be passed if the contents of the vector registers on return from the function are irrelevant.

Vector Count Register: The vector count register must contain the number of elements on which the function is to be performed. The vector count register is preserved across the call to the function.

Vector Mask Register: Each vector intrinsic elementary function has two entry points, one that is called with the vector mask mode off, and one that is called with the vector mask mode on.

When the mask mode is off, the entry name beginning with "V" in the following figures should be called. In this case, the vector mask mode will be off on return, and the vector mask register is undefined on return. If the entry name beginning with "W" is called, the vector mask mode must be on and the vector mask register will be preserved across the call. When the vector mask mode is on, the computations are performed only on those elements corresponding to true values in the vector mask register. The values corresponding to false values will be undefined on return from the function.

Vector Interruption Index: The vector interruption index should be set to 0 on calls to either type of entry. It will be 0 on return.

Program Mask

In some of the routines, fixed point overflow can occur during the calculations if the fixed point overflow exception is enabled. Initialization of the VS FORTRAN Version 2 run-time environment disables this exception.

Error Handling

If an element of the argument vector register is not within the argument range of the function, or if the result will underflow or overflow, the corresponding scalar routine will be called to perform the function. The scalar routine then issues the error message.

Vector Registers Used by Vector Intrinsic Elementary Functions

For each of the mathematical functions listed in column 1 of Figure 67 on page 354, there is a corresponding vector intrinsic elementary function that receives its arguments and returns its results in vector registers.

Those functions marked with an asterisk (*) take advantage of vector hardware to compute the results. The remaining functions facilitate vectorization in the compiler. In all cases, the vector intrinsic function returns the same results as would be obtained by passing the arguments to the scalar routine one by one. The argument ranges are identical to those of the scalar routines.

Each vector intrinsic elementary function has two entry points, one that is called with the vector mask mode off, and one that is called with the vector mask mode on.

Equivalent Scalar Function	Vector Entry If Mask Mode Off	Vector Entry If Mask Mode On	Intermediate Vector Registers Used
ACOS	V#ACOS	W#ACOS	
ASIN	V#ASIN	W#ASIN	
ATAN *	V#ATAN	W#ATAN	
ATAN2 *	V#ATAN2	W#ATAN2	
BTEST	V#BTEST	W#BTEST	
CABS *	VC#ABS	WC#ABS	2-3, 4-5
CCOS	VC#COS	WC#COS	
CDABS *	VCD#ABS	WCD#ABS	2-3, 4-5
CDCOS	VCD#COS	WCD#COS	
CDDVD#	VCDDVD#	WCDDVD#	
CDEXP	VCD#EXP	WCD#EXP	
CDLOG	VCD#LOG	WCD#LOG	
CDSIN	VCD#SIN	WCD#SIN	
CDSQRT	VCD#SQR	WCD#SQR	
CDVD#	VCDVD#	WCDVD#	
CEXP	VC#EXP	WC#EXP	
CLOG	VC#LOG	WC#LOG	
COS *	V#COS	W#COS	2-3
COSH	V#COSH	W#COSH	
COTAN	V#COTAN	W#COTAN	
CSIN	VC#SIN	WC#SIN	
CSQRT	VC#SQRT	WC#SQRT	
DACOS	VD#ACOS	WD#ACOS	
DASIN	VD#ASIN	WD#ASIN	
DATAN *	VD#ATAN	WD#ATAN	2-3, 4-5
DATAN2 *	VD#ATAN2	WD#ATAN2	2-3, 4-5
DCOS *	VD#COS	WD#COS	2-3, 4-5
DCOSH	VD#COSH	WD#COSH	
DCOTAN *	VD#COTN	WD#COTN	2-3, 4-5, 6-7
DERF	VD#ERF	WD#ERF	
DERFC	VD#ERFC	WD#ERFC	
DEXP *	VD#EXP	WD#EXP	2-3, 4-5
DGAMMA	VD#GAMMA	WD#GAMMA	
DLGAMA	VD#LGAMA	WD#LGAMA	
DLOG *	VD#LOG	WD#LOG	2-3, 4-5
DLOG10 *	VD#LOG10	WD#LOG10	2-3, 4-5
DSIN *	VD#SIN	WD#SIN	2-3, 4-5
DSINH	VD#SINH	WD#SINH	
DSQRT *	VD#SQRT	WD#SQRT	2-3
DTAN *	VD#TAN	WD#TAN	2-3, 4-5, 6-7
DTANH	VD#TANH	WD#TANH	
ERF	V#ERF	W#ERF	
ERFC	V#ERFC	W#ERFC	
EXP *	V#EXP	W#EXP	2-3
FCDCD#	VCDCD#	WCDCD#	
FCDXI#	VCDXI#	WCDXI#	
FCXPC#	VCXPC#	WCXPC#	
FCXPI#	VCXPI#	WCXPI#	
FDXPD# *	VDXPD#	WDXPD#	2-3, 4-5
FDXPI#	VDXPI#	WDXPI#	
FIXPI#	VIXPI#	WIXPI#	
FRXPI#	VRXPI#	WRXPI#	

Figure 67 (Part 1 of 2). Vector Entry Points and Intermediate Vector Registers Used

Equivalent Scalar Function	Vector Entry If Mask Mode Off	Vector Entry If Mask Mode On	Intermediate Vector Registers Used
FRXPR# *	VRXPR#	WRXPR#	2-3
GAMMA	V#GAMMA	W#GAMMA	
IBCLR	V#IBCLR	W#IBCLR	
IBSET	V#IBSET	W#IBSET	
LGAMMA	V#LGAMMA	W#LGAMMA	
LOG *	V#LOG	W#LOG	2-3
LOG10 *	V#LOG10	W#LOG10	2-3
SIN *	V#SIN	W#SIN	2-3
SINH	V#SINH	W#SINH	
SQRT *	V#SQRT	W#SQRT	2-3
TAN	V#TAN	W#TAN	
TANH	V#TANH	W#TANH	

Figure 67 (Part 2 of 2). Vector Entry Points and Intermediate Vector Registers Used

```
*       The following instructions show the use of a BALR
*       sequence to call the library vector square root
*       routine to compute the square root of 10 arguments.
*       All vector registers except 0, 1, 14, and 15 are
*       preserved across the call.
*       Vector mask mode is off.
        .
        .
        .
        VRCL   0              Clear vector interrupt index
        VSVMM  0              Set vector mask mode off
        VLVCA  10             Set vector count register
        LA     1,AMNT         Load address of argument array
        VLE    14,1           Load argument vector register
        IC     0,=X'FF'       Save scratch vector registers
        L      15,=V(V#SQRT)  Load address of entry point
        BALR   14,15          Call vector entry
        LA     1,ANSWER       Load address of result vector
        VSTE   0,1            Store results
        .
        .
        .
AMNT    DC     E'1',E'2',E'3',E'4',E'5'
        DC     E'6',E'7',E'8',E'9',E'10'
ANSWER  DC     10E'0'
        .
        .
        .
        DC     V(VFVIX#)
```

Figure 68. Assembler Language Calling Sequences with Vector Mask Mode Off

Notes to Figure 68:

1. The L and BALR sequence can be replaced by a CALL statement:

   `CALL  V#SQRT`

2. The code sequences will take the square root values, no matter what the value of the argument.

3. If an argument is invalid (for example, negative),the vector mathematical routine will pass the values to a special routine, which will then pass the arguments one by one to the scalar routine. The scalar routine will then identify the failing argument and issue a warning message. Results returned to the Assembler caller will be in vector register 0.

```
*       The following instructions show the use of a CALL
*       macro to call the vector library square root
*       routine to compute the square roots of the
*       nonnegative elements of a vector of length 10.
*       All vector registers except 0, 1, 14, and 15 are
*       preserved across the call.  Vector mask mode is on.
        .
        .
        .
        VRCL  0            Clear vector interrupt index
        VSVMM 1            Set vector mask mode on
        VLVCA 10           Set vector count register
        LA    1,AMNT       Load address of array
        VLE   14,1         Load argument vector register
        SER   0,0          Set comparand for vector compare
        VCEQ  4,0,14       Set vector mask register to 1 if
*                          array elements are nonnegative
        IC    0,X'00'      Don't save scratch registers
        CALL  W#SQRT       Call vector routine
        LA    1,ANSWER     Load address of result vector
        VSTME 0,1          Store square roots of
*                          nonnegative elements only
        VSVMM 0            Reset vector mask mode off
        .
        .
AMNT    DC    E'1',E'-2',E'3',E'-4',E'5'
        DC    E'-6',E'7',E'-8',E'9',E'-10'
ANSWER  DC    10E'0'
        .
        .
```

Figure 69. Assembler Language Calling Sequences with Vector Mask Mode On

Notes to Figure 69:

1. The CALL statement can be replaced by a BALR sequence:

```
L     15,=V(W#SQRT)
BALR  14,15
```

2. The code sequence sets the vector mask register to indicate which arguments are greater than or equal to zero. Only those arguments are processed.

Appendix C. Sample Storage Printouts

Output from Symbolic Dumps

SDUMP output is produced, upon abnormal termination of your program, if the program units were compiled with the TEST option or without the NOSDUMP option and if the run-time option ABSDUMP was specified. The SDUMP output contains information on all variables and arrays in each program unit on the save area chain, as well as those in the program currently being processed.

Output Format

In general, the output shows variable items (one line only) and array (more than one line) items. variable and array items can contain either character or noncharacter data, but not both.

In addition, variable and array items both identify valid variable types (shown as yyy in the formats).

Variable Noncharacter

The variable value printing scheme for noncharacter data is as follows:

```
xxxxxxxxxxxxxxxxxx yyy zzzzzzzz
```

where:

xxxxxxxxxxxxxxxxxx is the variable name area.

If the variable name is longer than 18 characters, the name is printed on a separate line above the type and data information.

yyy is the data type.

zzzzzzzz is the area for the formatted output.

The data type field (yyy) can have one of the following values:

Code	**Represents**	**Length**
I4	Integer	(4 bytes)
I2	Integer	(2 bytes)
L4	Logical	(4 bytes)
L1	Logical	(1 byte)
R4	Real	(4 bytes)
R8	Real	(8 bytes)
R16	Real	(16 bytes)
C8	Complex	(8 bytes, 4+4)
C16	Complex	(16 bytes, 8+8)
C32	Complex	(32 bytes, 16+16)

Figure 70. Valid Values for Data Type Field in SDUMP Output

Variable Character

The variable value printing scheme for character data is as follows:

```
xxxxxxxxxxxxxxxxxx CHR
```

where:

xxxxxxxxxxxxxxxxxx is the variable name.

If the variable name is longer than 18 characters, the name is printed on a separate line above the type information.

Character Data Format

The character data format is as follows:

```
xxxxxxxx   aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa *bbbbbbbbbbbbbbbb*
```

where:

xxxxxxxx. is the count of the next character displayed—the value is the decimal number of the character.

aaaaaaaa is the hexadecimal representation of up to 4 bytes of character data—as many aa's are used as are needed to display the internal form of the data.

bbbbbbbbbbbbbbbb is the EBCDIC representation of up to 16 bytes of character data—as many b's are used as are needed to display the data.

Note: Unprintable characters are translated to the character period (.); asterisks (*) are the delimiters of the EBCDIC character area.

Array

The array value printing scheme is as follows:

```
ARRAY: xxxxxxxx   TYPE:yyy
```

where:

xxxxxxxx is the array name.

yyy is the data type.
Valid values for yyy are listed in Figure 70 on page 357.

Array Specification

```
DIMENSION x:  (yyyyyyyy:zzzzzzzz)aaa
```

where:

x is the dimension (from 1 to 7).

yyyyyyyy is the lower bound.

zzzzzzzz is the upper bound.

aaa is either blank or * ASSUMED SIZE ARRAY*.

Note: * ASSUMED SIZE ARRAY* appears only for the last dimension—there is one dimension line for each dimension of the array.

Array Contents

For the display of the contents of an array, the output is divided into two parts: Part 1 describes the array name and the current element indexes, and part 2 displays the contents of the array.

The following shows how part 1 is formatted:

```
xxxxxxxx(dim1,dim2,dim3,dim4,dim5,dim6,dim7)
```

where:

xxxxxxxx is the array name.

dim1-dim7 are the indexes of the element value. The last dimension for the array is displayed by the special character, #. This line is printed whenever the previous dimension changes or every 50 lines.

The second part of the output describes the contents of the array. The data line has the following format:

```
# = nnnn data
```

where:

nnnn is the last dimension index value for the array element

data is the data value for the array element.

A line of hyphens in the output marks the end of output for each variable or array item (*not* an array element).

Array Message

The following message is issued if some array elements are missing from the printed dump output:

```
ARRAY ELEMENTS WITH A VALUE OF ZERO, BLANK, OR FALSE ARE NOT
PRINTED.
```

Control Flow Information

The following shows the printing scheme of the portion of symbolic dump output that indicates where a call originated and what other routines the program calls, if applicable:

```
MODULE xxxxxxxx WAS CALLED BY yyyyyyyy.

OP/SYS message fragment for OPERATING SYSTEM.

FROM OFFSET aaaaaa AT ISN. NO. bbbbbbbbbb.
```

wnere:

xxxxxxxx identifies the caller module.

yyyyyyyy identifies the called routine.

OP/SYS is the operating system: DOS, MVS, VS1, or CMS.

aaaaaa is the offset into the program unit. If blanks appear, then the offset is not available.

bbbbbbbbbb is the internal statement number (ISN). If double asterisks appear, the ISN information is unavailable.

Note: The message fragment is used in conjunction with other fragments to identify the CALLs and RETURNs of the program units.

```
MODULE xxxxxxxx LAST CALLED yyyyyyyy
```

where:

xxxxxxxx is the calling module name.

yyyyyyyy is the called module name.

Note: The message fragment is used in tracing the control flow of program units.

```
MODULE xxxxxxxx DID NOT CALL ANY OTHER ROUTINES.
```

where:

xxxxxxxx is the routine that did not call any other routines.

Note: The message fragment completes the group of fragments identifying the control flow scheme.

I/O Unit Information

The following messages appear only for post-ABEND processing (VPOST or VPOSA):

1. Default units

   DATA SET REFERENCE NUMBER TABLE. NUMBER OF ENTRIES IS xxx.
   Indicates the number of units available to the FORTRAN program is xxx.

   DEFAULT UNIT FOR THE PRINTER IS xxx.
   Indicates the default output device is xxx.

   DEFAULT UNIT FOR THE READER IS xxx.
   Indicates the default input device is xxx.

   DEFAULT UNIT FOR THE PUNCH IS xxx.
   Indicates the default punch output device is xxx.

   DEFAULT UNIT FOR THE OBJECT TIME ERROR MESSAGES IS xxx.
   Indicates that error messages issued by the FORTRAN program will go to unit xxx. This includes messages issued by AFBVPOSA (abnormal termination) or AFBSDUMP (SDUMP).

2. Active units

   FILE ON UNIT xxx IS ACTIVE.
   Indicates that input/output activity has been proceeding on unit xxx.

3. Inactive (or formerly used) units

   FILE IS INACTIVE. LAST CONNECTED UNIT IS xxx.
   Indicates that file on unit xxx has been the object of a CLOSE or REWIND statement.

I/O Unit Status Information

The following message fragments describe the identified unit. The messages may not appear in this sequence, and not all may appear.

```
FILE IS USED FOR ASYNCHRONOUS SEQUENTIAL I/O.
FILE IS USED FOR SYNCHRONOUS SEQUENTIAL I/O.
FILE IS USED FOR DIRECT I/O.
FILE USES VSAM ACCESS METHOD.
FILE IS NAMED.
FILE STATUS IS OLD.
FILE STATUS IS UNKNOWN.
FILE IS FORMATTED.
FILE IS UNFORMATTED.
FILE HAS PERMANENT OPEN ERROR.
FILE HAS HAD FIRST I/O ERROR.
FILE NAME USED IS xxxxxxxx.
```

where xxxxxxxx is the ddname.

Examples of Sample Programs and Symbolic Dump Output

The following are three examples of sample programs and symbolic dump output. The first two examples show the two types of output, the first for variable items and the second for array items. Be aware, however, that if your program assigns values to both variable and array items, the output formats will be mixed. The third example is a sample of what you might get after a non-recoverable failure.

Example 1. Variable Items

This program assigns values to variable items.

```
@PROCESS
C
C     SAMPLE PROGRAM TO DEMONSTRATE SDUMP OF SCALAR VARIABLES
C
C
C     SPECIFY THE VARIABLE TYPES
C
      COMPLEX*8 C2,C1
      COMPLEX*16 C3
      COMPLEX*32 C4
      CHARACTER CH1,CH2*8
      REAL*8 A,B
      REAL*16 YYYY,ZZZZ
      INTEGER*2 JJJ,KKK,OOO,PPP
      LOGICAL*1 P
      LOGICAL*4 Q
C
C     ASSIGN THE VALUES
C
      P=.TRUE.
      Q=.FALSE.
      CH2='ABCDEFGH'
      CH1='1'
      R=32.3733
      A = 5.5
      NUMLTS=7
      IABLSE=6
      ZZZZ=4.0E5
      IIII=1111
      JJJ=222
      OOO=32767
      PPP=-32768
      B=222.222
      KKK=999
      LLL=121212
      MMM = 2147483647
      NNN = -2147483647
      YYYY=25252525
      C1=(1.,1.)
      C2=(2.,2.)
      C3=(3.D0,3.D0)
20    C4=(4.Q0,4.Q0)
C
C     PRINT MESSAGE AND INVOKE SDUMP
C
C     WRITE(6,*) ' CALL SDUMP WITH SCALAR VARIABLES OF VARIOUS TYPES '
      CALL SDUMP
10    STOP
      END
```

Figure 71. Example 1—Source Program

Example 1. Output

The output is as follows:

```
1SDUMP - SYMBOLIC DUMP FOR MODULE: MAIN
0 MODULE MAIN     WAS CALLED BY OP/SYS  .
0 MODULE MAIN     LAST CALLED AFBSDUMQ.
  FROM OFFSET 000438 AT ISN. NO.        33.
0NNN                 I4   -2147483647
 ---------------------------------------------------------------------------------
 MMM                 I4    2147483647
 ---------------------------------------------------------------------------------
 LLL                 I4        121212
 ---------------------------------------------------------------------------------
 IIII                I4          1111
 ---------------------------------------------------------------------------------
 IABLSE              I4             6
 ---------------------------------------------------------------------------------
 NUMLTS              I4             7
 ---------------------------------------------------------------------------------
 R                   R4    0.323733E+02
 ---------------------------------------------------------------------------------
 Q                   L4   F
 ---------------------------------------------------------------------------------
 P                   L1   T
 ---------------------------------------------------------------------------------
 PPP                 I2   -32768
 ---------------------------------------------------------------------------------
 OOO                 I2    32767
 ---------------------------------------------------------------------------------
 KKK                 I2      999
 ---------------------------------------------------------------------------------
 JJJ                 I2      222
 ---------------------------------------------------------------------------------
 ZZZZ                R16   0.40000000000000000000000000000000Q+06
 ---------------------------------------------------------------------------------
 YYYY                R16   0.25252525000000000000000000000000Q+08
 ---------------------------------------------------------------------------------
 B                   R8    0.22222200012207D+03
 ---------------------------------------------------------------------------------
 A                   R8    0.55000000000000D+01
 ---------------------------------------------------------------------------------
 CH2                 CHR
        1                 C1C2C3C4 C5C6C7C8                      *ABCDEFGH        *
 ---------------------------------------------------------------------------------
 CH1                 CHR
        1                 F1                                     *1               *
 ---------------------------------------------------------------------------------
 C4                  C32   0.40000000000000000000000000000000Q+01  0.40000000000000000000000000000000Q+01
 ---------------------------------------------------------------------------------
 C3                  C16   0.30000000000000D+01  0.30000000000000D+01
 ---------------------------------------------------------------------------------
 C1                  C8    0.100000E+01  0.100000E+01
 ---------------------------------------------------------------------------------
 C2                  C8    0.200000E+01  0.200000E+01
 ---------------------------------------------------------------------------------
 END OF SYMBOL DUMP PROCESSING FOR MAIN    .
```

Figure 72. Example 1—Output

Example 2. Array Items

This program assigns values to array items.

```
@PROCESS                                                                00000100
C                                                                       00000200
C     SAMPLE PROGRAM TO DEMONSTRATE SDUMP OF ARRAY VARIABLES            00000300
C                                                                       00000400
C                                                                       00000500
C     SPECIFY THE VARIABLE TYPES                                        00000600
C                                                                       00000700
      COMPLEX*8 C2(5),C1(5)                                             00000800
      COMPLEX*16 C3(5)                                                  00000900
      COMPLEX*32 C4(5)                                                  00001000
      CHARACTER CH1(5),CH2(5)*8                                         00001100
      REAL*8 A(5),B(5)                                                  00001200
      REAL*16 YYYY(5),ZZZZ(5)                                           00001300
      INTEGER*2 JJJ(5),KKK(5)                                           00001400
      INTEGER NUMLTS(5),IABLSE(5),IIII(5),LLL(5)                        00001500
      LOGICAL*1 P(5)                                                    00001600
      LOGICAL*4 Q(5)                                                    00001700
C                                                                       00001800
C     ASSIGN THE VALUES                                                 00001900
C                                                                       00002000
      P(1)=.TRUE.                                                       00002100
      P(2)=.FALSE.                                                      00002200
      P(3)=.TRUE.                                                       00002300
      P(4)=.FALSE.                                                      00002400
      P(5)=.TRUE.                                                       00002500
      Q(1)=.TRUE.                                                       00002600
      Q(2)=.FALSE.                                                      00002700
      Q(3)=.FALSE.                                                      00002800
      Q(4)=.FALSE.                                                      00002900
      Q(5)=.TRUE.                                                       00003000
      CH2(1)(:)='ABCDEFGH'                                              00003100
      CH2(2)(:)='ABCIJKLM'                                              00003200
      CH2(3)(:)='ABCNOPQR'                                              00003300
      CH2(4)(:)='ABCSTUVW'                                              00003400
      CH2(5)(:)='ABCXYZAB'                                              00003500
      CH1(1)(:)='1'                                                     00003600
      CH1(2)(:)='2'                                                     00003700
      CH1(3)(:)='3'                                                     00003800
      CH1(4)(:)='4'                                                     00003900
      CH1(5)(:)='5'                                                     00004000
      A(1) = 5.5                                                        00004100
      A(2) = 4.5                                                        00004200
      A(3) = 3.5                                                        00004300
      A(4) = 2.5                                                        00004400
      A(5) = 1.5                                                        00004500
      NUMLTS(1)=7                                                       00004600
      NUMLTS(2)=6                                                       00004700
      NUMLTS(3)=5                                                       00004800
      NUMLTS(4)=4                                                       00004900
      NUMLTS(5)=3                                                       00005000
      IABLSE(1)=6                                                       00005100
      IABLSE(2)=7                                                       00005200
      IABLSE(3)=8                                                       00005300
      IABLSE(4)=9                                                       00005400
      IABLSE(5)=10                                                      00005500
```

Figure 73 (Part 1 of 2). Example 2—Source Program

```
      ZZZZ(1)=4.0E5
      ZZZZ(2)=4.0E3
      ZZZZ(3)=4.0E2
      ZZZZ(4)=4.0E1
      ZZZZ(5)=4.0E0
      IIII(1)=1111
      IIII(2)=3211
      IIII(3)=4311
      IIII(4)=6511
      IIII(5)=1541
      JJJ(1)=212
      JJJ(2)=242
      JJJ(3)=232
      JJJ(4)=252
      JJJ(5)=262
      B(1)=111.222
      B(2)=222.222
      B(3)=333.222
      B(4)=444.222
      B(5)=555.222
      KKK(1)=899
      KKK(2)=799
      KKK(3)=699
      KKK(4)=599
      KKK(5)=499
      LLL(1)=212
      LLL(2)=312
      LLL(3)=412
      LLL(4)=512
      LLL(5)=612
      YYYY(1)=15151515
      YYYY(2)=25252525
      YYYY(3)=35353535
      YYYY(4)=45454545
      YYYY(5)=55555555
      C1(1)=(5.,1.)
      C1(2)=(4.,2.)
      C1(3)=(3.,3.)
      C1(4)=(2.,4.)
      C1(5)=(1.,5.)
      C2(1)=(2.,10.)
      C2(2)=(4.,8.)
      C2(3)=(6.,6.)
      C2(4)=(8.,4.)
      C2(5)=(10.,2.)
      C3(1)=(3.D0,13.D0)
      C3(2)=(6.D0,11.D0)
      C3(3)=(9.D0,9.D0)
      C3(4)=(12.D0,7.D0)
      C3(5)=(15.D0,5.D0)
      C4(1)=(4.Q0,4.Q0)
      C4(2)=(3.Q0,5.Q0)
      C4(3)=(2.Q0,6.Q0)
      C4(4)=(1.Q0,7.Q0)
      C4(5)=(0.Q0,8.Q0)
C
C     PRINT MESSAGE AND INVOKE SDUHP
C
      WRITE(6,*) ' CALL SDUHP WITH ARRAY VARIABLES OF VARIOUS TYPES '
      CALL SDUHP
      STOP
      END
```

Figure 73 (Part 2 of 2). Example 2—Source Program

Example 2. Output

The output is as follows:

```
  CALL SDUMP WITH ARRAY VARIABLES OF VARIOUS TYPES

1SDUMP - SYMBOLIC DUMP FOR MODULE: MAIN
0 MODULE MAIN     WAS CALLED BY OP/SYS  .
0 MODULE MAIN     LAST CALLED AFBSDUMQ.
  FROM OFFSET 000CF8 AT ISN. NO.        102.
0ARRAY:  Q   TYPE:L4
0 DIMENSION 1: (       1:       5)
   Q( #)
     # = 1       T
     # = 2       F
     # = 3       F
     # = 4       F
     # = 5       T
 ---------------------------------------------------------------------------
0ARRAY:  P   TYPE:L1
0 DIMENSION 1: (       1:       5)
   P( #)
     # = 1       T
     # = 2       F
     # = 3       T
     # = 4       F
     # = 5       T
 ---------------------------------------------------------------------------
0ARRAY:  LLL   TYPE:I4
0 DIMENSION 1: (       1:       5)
   LLL( #)
     # = 1             212
     # = 2             312
     # = 3             412
     # = 4             512
     # = 5             612
 ---------------------------------------------------------------------------
0ARRAY:  IIII   TYPE:I4
0 DIMENSION 1: (       1:       5)
   IIII( #)
     # = 1            1111
     # = 2            3211
     # = 3            4311
     # = 4            6511
     # = 5            1541
 ---------------------------------------------------------------------------
0ARRAY:  IABLSE   TYPE:I4
0 DIMENSION 1: (       1:       5)
   IABLSE( #)
     # = 1               6
     # = 2               7
     # = 3               8
     # = 4               9
     # = 5              10
 ---------------------------------------------------------------------------
0ARRAY:  NUMLTS   TYPE:I4
0 DIMENSION 1: (       1:       5)
   NUMLTS( #)
     # = 1               7
     # = 2               6
     # = 3               5
     # = 4               4
     # = 5               3
 ---------------------------------------------------------------------------
```

Figure 74 (Part 1 of 4). Example 2—Output

```
OARRAY:  KKK   TYPE:I2
0  DIMENSION 1: (       1:       5)
   KKK( #)
     # = 1          899
     # = 2          799
     # = 3          699
     # = 4          599
     # = 5          499
 -----------------------------------------------------------------------------
OARRAY:  JJJ   TYPE:I2
0  DIMENSION 1: (       1:       5)
   JJJ( #)
     # = 1          212
     # = 2          242
     # = 3          232
     # = 4          252
     # = 5          262
 -----------------------------------------------------------------------------
OARRAY:  ZZZZ   TYPE:R16
0  DIMENSION 1: (       1:       5)
   ZZZZ( #)
     # = 1        0.400000000000000000000000000000000Q+06
     # = 2        0.400000000000000000000000000000000Q+04
     # = 3        0.400000000000000000000000000000000Q+03
     # = 4        0.400000000000000000000000000000000Q+02
     # = 5        0.400000000000000000000000000000000Q+01
 -----------------------------------------------------------------------------
OARRAY:  YYYY   TYPE:R16
0  DIMENSION 1: (       1:       5)
   YYYY( #)
     # = 1        0.151515150000000000000000000000000Q+08
     # = 2        0.252525250000000000000000000000000Q+08
     # = 3        0.353535350000000000000000000000000Q+08
     # = 4        0.454545450000000000000000000000000Q+08
     # = 5        0.555555550000000000000000000000000Q+08
 -----------------------------------------------------------------------------
OARRAY:  B   TYPE:R8
0  DIMENSION 1: (       1:       5)
   B( #)
     # = 1        0.11122200012207D+03
     # = 2        0.22222200012207D+03
     # = 3        0.33322192382813D+03
     # = 4        0.44422192382813D+03
     # = 5        0.55522192382813D+03
 -----------------------------------------------------------------------------
OARRAY:  A   TYPE:R8
0  DIMENSION 1: (       1:       5)
   A( #)
     # = 1        0.55000000000000D+01
     # = 2        0.45000000000000D+01
     # = 3        0.35000000000000D+01
     # = 4        0.25000000000000D+01
     # = 5        0.15000000000000D+01
 -----------------------------------------------------------------------------
```

Figure 74 (Part 2 of 4). Example 2—Output

```
0ARRAY:  CH2   TYPE:CHR
0 DIMENSION 1: (       1:       5)
   CH2( #)
     # = 1
         1     C1C2C3C4 C5C6C7C8                       *ABCDEFGH       *
     # = 2
         1     C1C2C3C9 D1D2D3D4                       *ABCIJKLM       *
     # = 3
         1     C1C2C3D5 D6D7D8D9                       *ABCNOPQR       *
     # = 4
         1     C1C2C3E2 E3E4E5E6                       *ABCSTUVW       *
     # = 5
         1     C1C2C3E7 E8E9C1C2                       *ABCXYZAB       *
 ---------------------------------------------------------------------------
0ARRAY:  CH1   TYPE:CHR
0 DIMENSION 1: (       1:       5)
   CH1( #)
     # = 1
         1     F1                                      *1              *
     # = 2
         1     F2                                      *2              *
     # = 3
         1     F3                                      *3              *
     # = 4
         1     F4                                      *4              *
     # = 5
         1     F5                                      *5              *
 ---------------------------------------------------------------------------
0ARRAY:  C4   TYPE:C32
0 DIMENSION 1: (       1:       5)
   C4( #)
     # = 1        0.40000000000000000000000000000000Q+01  0.40000000000000000000000000000000Q+01
     # = 2        0.30000000000000000000000000000000Q+01  0.50000000000000000000000000000000Q+01
     # = 3        0.20000000000000000000000000000000Q+01  0.60000000000000000000000000000000Q+01
     # = 4        0.10000000000000000000000000000000Q+01  0.70000000000000000000000000000000Q+01
     # = 5        0.00000000000000000000000000000000Q+00  0.80000000000000000000000000000000Q+01
 ---------------------------------------------------------------------------
0ARRAY:  C3   TYPE:C16
0 DIMENSION 1: (       1:       5)
   C3( #)
     # = 1        0.30000000000000D+01  0.13000000000000D+02
     # = 2        0.60000000000000D+01  0.11000000000000D+02
     # = 3        0.90000000000000D+01  0.90000000000000D+01
     # = 4        0.12000000000000D+02  0.70000000000000D+01
     # = 5        0.15000000000000D+02  0.50000000000000D+01
 ---------------------------------------------------------------------------
0ARRAY:  C1   TYPE:C8
0 DIMENSION 1: (       1:       5)
   C1( #)
     # = 1        0.500000E+01  0.100000E+01
     # = 2        0.400000E+01  0.200000E+01
     # = 3        0.300000E+01  0.300000E+01
     # = 4        0.200000E+01  0.400000E+01
     # = 5        0.100000E+01  0.500000E+01
```

Figure 74 (Part 3 of 4). Example 2—Output

```
 ---------------------------------------------------------------------------
0ARRAY:  C2   TYPE:C8
0 DIMENSION 1: (        1:        5)
   C2( #)
     # = 1         0.200000E+01  0.100000E+02
     # = 2         0.400000E+01  0.800000E+01
     # = 3         0.600000E+01  0.600000E+01
     # = 4         0.800000E+01  0.400000E+01
     # = 5         0.100000E+02  0.200000E+01
 ---------------------------------------------------------------------------
 END OF SYMBOL DUMP PROCESSING FOR MAIN
```

Figure 74 (Part 4 of 4). Example 2—Output

Example 3. Non-recoverable Failure

This program will attempt to store data, but will fail because the index into the array has a number that is too large, and the program attempts to store the array in an area that doesn't belong to the program.

```
DIMENSION A(10)                                         00000900
A(5)=3.2                                                00001000
I=99999999                                              00001400
A(I)=2.3                                                00001800
STOP                                                    00001900
END                                                     00002000
```

Figure 75. Example 3—Source Program

The output you get will vary, depending on the compiler options GOSTMT/NOGOSTMT, SDUMP/NOSDUMP, and TEST/NOTEST, as follows:

- The lines of the AFB240I message following the register contents are option dependent. For details on AFB240I, see Appendix D, "Library Procedures and Messages" on page 375.
- Traceback information appears next and is dependent on the GOSTMT option. When the traceback includes one or more subprograms compiled with GOSTMT, ISNs appear in those lines for those programs; otherwise, ** appears. For details on traceback, see *VS FORTRAN Version 2 Programming Guide*.
- I/O unit and unit status information appears next and is option independent. This unit information is produced only if the program abnormally terminates. For details on unit status, see "I/O Unit Information" on page 360.
- Control flow information appears last and is dependent on the SDUMP or TEST option. For a program unit active at abend that was compiled with SDUMP or TEST, the control flow information contains ISNs/line numbers; otherwise, it contains ** in those fields. For details on control flow, see "Control Flow Information" on page 359.

To get post-abend data, the object error unit must be directed to a disk or SYSOUT file. No output will be sent to the object error unit if it is directed to a terminal device.

Sample output from running this program follows. The first sample output is obtained when the program is run without any any run-time options specified.

The second sample output is obtained when the program is run with ABMODLST specified. The third sample output is obtained when the program is run with ABMODLST and ABSDUMP specified.

Example 3. Output

This first sample output is obtained when the program is run without any any run-time options specified.

```
AFB210I VFNTH : PROGRAM INTERRUPT - ADDRESSING EXCEPTION
        VFNTH : PSW FFE400059202020C
        VFNTH : LAST EXECUTED FORTRAN STATEMENT IN PROGRAM MAIN AT ISN 4 (OFFSET 00020C).

TRACEBACK OF CALLING ROUTINES; MODULE ENTRY ADDRESS = 020000.
------------------------------------------------------------------------------
 MAIN (020000) CALLED BY OPERATING SYSTEM.
------------------------------------------------------------------------------

STANDARD CORRECTIVE ACTION TAKEN. EXECUTION CONTINUING.

AFB240I VABEX : ABEND CODE IS: SYSTEM 0C5, USER 0.
        VABEX : PSW=FFE400059202020C ENTRY POINT=020000.
        VABEX : REGS 0 - 3 05F5E0FF 00000000 0002021C 000200D8
        VABEX : REGS 4 - 7 00F61362 00000006 17D783FC 0B000850
        VABEX : REGS 8 -11 40F79BB8 00F7ABB8 E3404040 00F79BB8
        VABEX : REGS 12-15 00020000 000200D8 40020226 000201EC
        VABEX : FRGS 0 & 2 4124CCCD 00000000 00000000 00000000
        VABEX : FRGS 4 & 6 00000000 00000000 00000000 00000000
        VABEX : ABEND IN MODULE MAIN AT ISN 4 (OFFSET 0000020C).
DMSABN155T USER ABEND 0240 CALLED FROM 02F0D8.
```

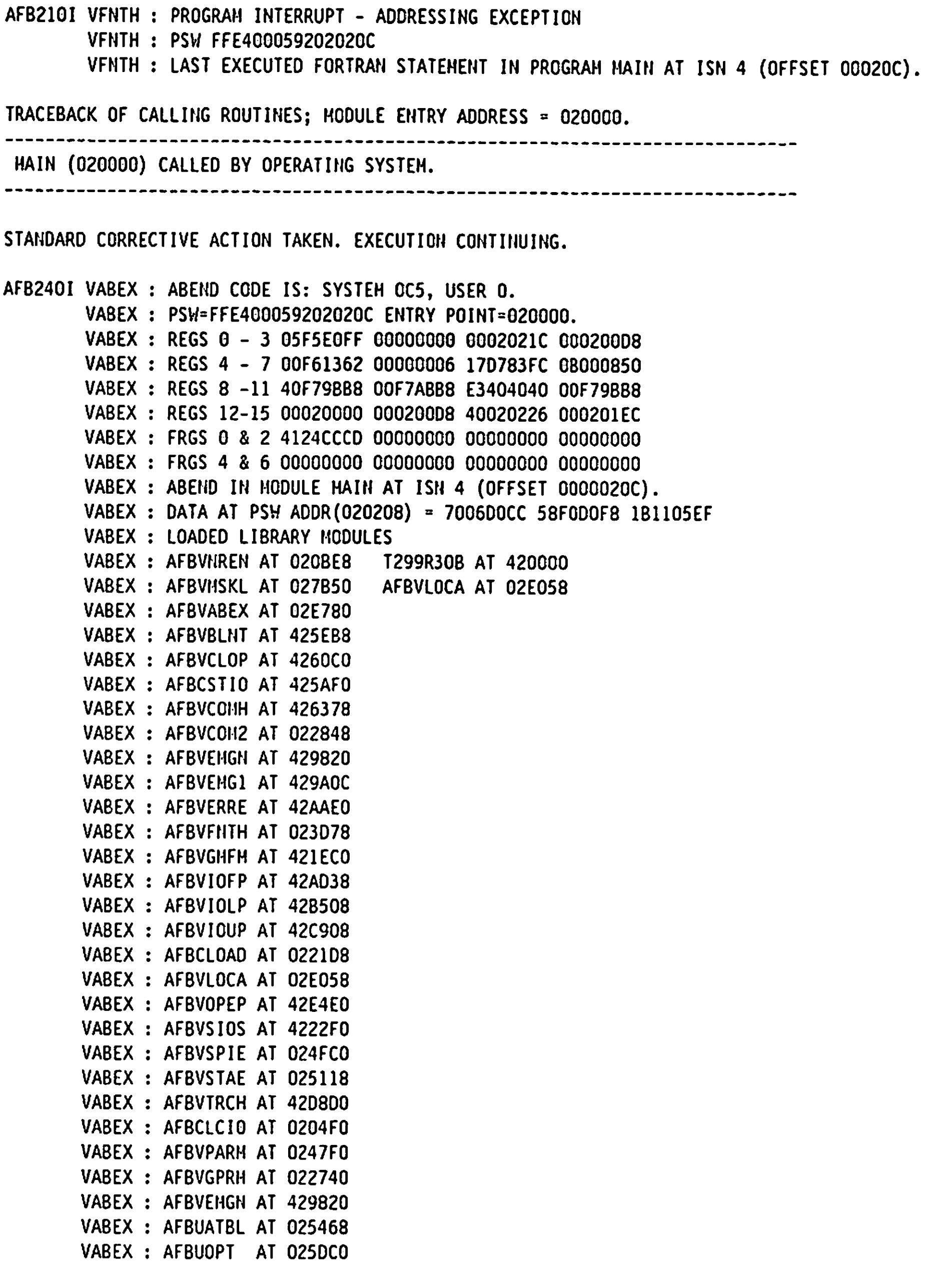
Figure 76. Example 3—Output Without Run-Time Options

This second sample output is obtained when the program is run with run-time option ABMODLST specified.

```
AFB210I VFNTH : PROGRAM INTERRUPT - ADDRESSING EXCEPTION
        VFNTH : PSW FFE400059202020C
        VFNTH : LAST EXECUTED FORTRAN STATEMENT IN PROGRAM MAIN AT ISN 4 (OFFSET 00020C).

TRACEBACK OF CALLING ROUTINES; MODULE ENTRY ADDRESS = 020000.
---------------------------------------------------------------------
 MAIN (020000) CALLED BY OPERATING SYSTEM.
---------------------------------------------------------------------

STANDARD CORRECTIVE ACTION TAKEN. EXECUTION CONTINUING.

AFB240I VABEX : ABEND CODE IS: SYSTEM OC5, USER 0.
        VABEX : PSW=FFE400059202020C ENTRY POINT=020000.
        VABEX : REGS 0 - 3 05F5E0FF 00000000 0002021C 000200D8
        VABEX : REGS 4 - 7 00F61362 00000006 17D783FC 0B000850
        VABEX : REGS 8 -11 40F79BB8 00F7ABB8 E3404040 00F79BB8
        VABEX : REGS 12-15 00020000 000200D8 40020226 000201EC
        VABEX : FRGS 0 & 2 4124CCCD 00000000 00000000 00000000
        VABEX : FRGS 4 & 6 00000000 00000000 00000000 00000000
        VABEX : ABEND IN MODULE MAIN AT ISN 4 (OFFSET 0000020C).
        VABEX : DATA AT PSW ADDR(020208) = 7006D0CC 58F0D0F8 1B1105EF
        VABEX : LOADED LIBRARY MODULES
        VABEX : AFBVNREN AT 020BE8   T299R30B AT 420000
        VABEX : AFBVMSKL AT 027B50   AFBVLOCA AT 02E058
        VABEX : AFBVABEX AT 02E780
        VABEX : AFBVBLNT AT 425EB8
        VABEX : AFBVCLOP AT 4260C0
        VABEX : AFBCSTIO AT 425AF0
        VABEX : AFBVCOMH AT 426378
        VABEX : AFBVCOM2 AT 022848
        VABEX : AFBVEMGN AT 429820
        VABEX : AFBVEMG1 AT 429A0C
        VABEX : AFBVERRE AT 42AAE0
        VABEX : AFBVFNTH AT 023D78
        VABEX : AFBVGMFM AT 421EC0
        VABEX : AFBVIOFP AT 42AD38
        VABEX : AFBVIOLP AT 42B508
        VABEX : AFBVIOUP AT 42C908
        VABEX : AFBCLOAD AT 0221D8
        VABEX : AFBVLOCA AT 02E058
        VABEX : AFBVOPEP AT 42E4E0
        VABEX : AFBVSIOS AT 4222F0
        VABEX : AFBVSPIE AT 024FC0
        VABEX : AFBVSTAE AT 025118
        VABEX : AFBVTRCH AT 42D8D0
        VABEX : AFBCLCIO AT 0204F0
        VABEX : AFBVPARM AT 0247F0
        VABEX : AFBVGPRM AT 022740
        VABEX : AFBVEMGN AT 429820
        VABEX : AFBUATBL AT 025468
        VABEX : AFBUOPT  AT 025DC0
DHSABN155T USER ABEND 0240 CALLED FROM 02F0D8.
```

Figure 77. Example 3—Output for ABMODLST

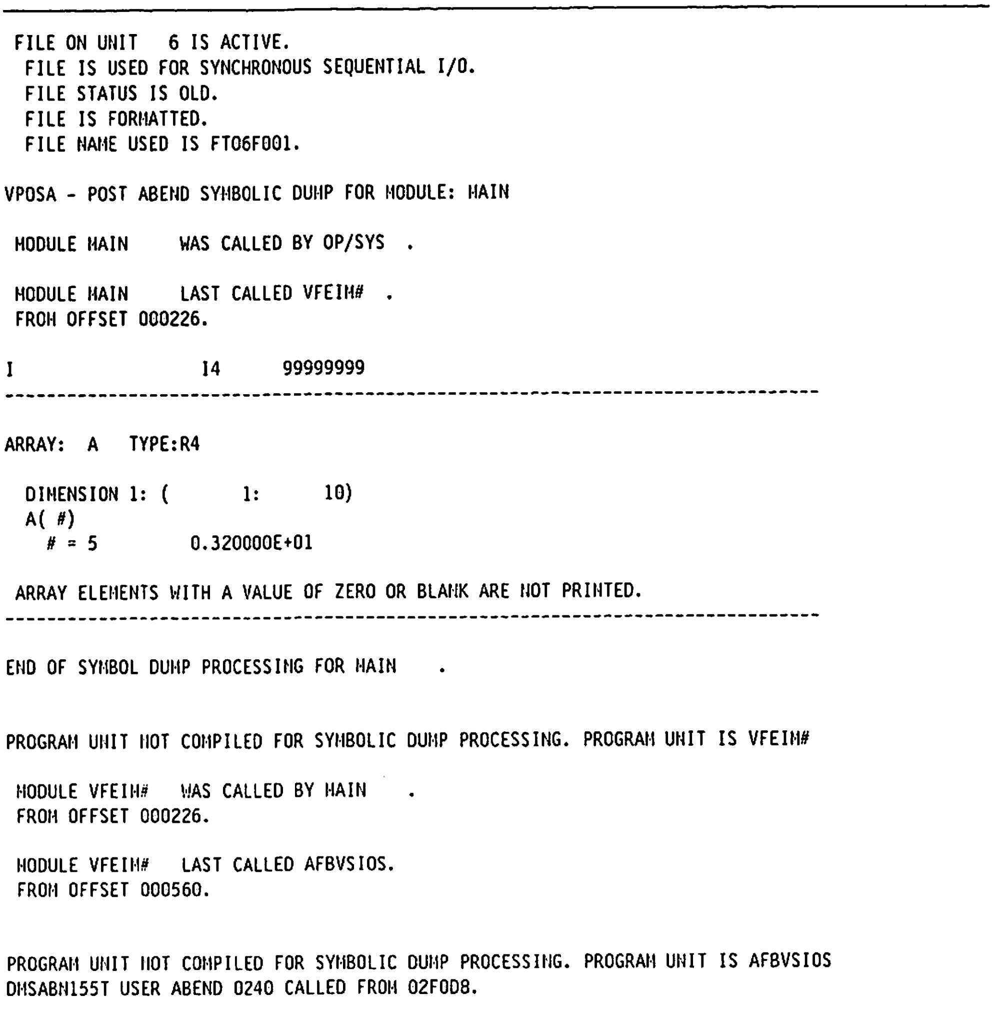
This third sample output is obtained when the program is run with run-time options ABMODLST and ABSDUMP specified.



```
AFB210I VFNTH : PROGRAM INTERRUPT - ADDRESSING EXCEPTION
        VFNTH : PSW FFE400059202020C
        VFNTH : LAST EXECUTED FORTRAN STATEMENT IN PROGRAM MAIN AT ISN 4 (OFFSET 00020C).

TRACEBACK OF CALLING ROUTINES; MODULE ENTRY ADDRESS = 020000.
------------------------------------------------------------------------
 MAIN (020000) CALLED BY OPERATING SYSTEM.
------------------------------------------------------------------------

STANDARD CORRECTIVE ACTION TAKEN. EXECUTION CONTINUING.

AFB240I VABEX : ABEND CODE IS: SYSTEM 0C5, USER 0.
        VABEX : PSW=FFE400059202020C ENTRY POINT=020000.
        VABEX : REGS 0 - 3 05F5E0FF 00000000 0002021C 000200D8
        VABEX : REGS 4 - 7 00F61362 00000006 17D783FC 0B000850
        VABEX : REGS 8 -11 40F79BB8 00F7ABB8 E3404040 00F79BB8
        VABEX : REGS 12-15 00020000 000200D8 40020226 000201EC
        VABEX : FRGS 0 & 2 4124CCCD 00000000 00000000 00000000
        VABEX : FRGS 4 & 6 00000000 00000000 00000000 00000000
        VABEX : ABEND IN MODULE MAIN AT ISN 4 (OFFSET 0000020C).
        VABEX : DATA AT PSW ADDR(020208) = 7006D0CC 58F0D0F8 1B1105EF
        VABEX : LOADED LIBRARY MODULES
        VABEX : AFBVNREN AT 020BE8   T299R30B AT 420000
        VABEX : AFBVMSKL AT 027B50   AFBVLOCA AT 02E058
        VABEX : AFBVABEX AT 02E780
        VABEX : AFBVBLNT AT 425EB8
        VABEX : AFBVCLOP AT 4260C0
        VABEX : AFBCSTIO AT 425AF0
        VABEX : AFBVCOMM AT 426378
        VABEX : AFBVCOM2 AT 022848
        VABEX : AFBVEMGN AT 429820
        VABEX : AFBVEMG1 AT 429A0C
        VABEX : AFBVERRE AT 42AAE0
        VABEX : AFBVFNTH AT 023D78
        VABEX : AFBVGMFM AT 421EC0
        VABEX : AFBVIOFP AT 42AD38
        VABEX : AFBVIOLP AT 42B508
        VABEX : AFBVIOUP AT 42C908
        VABEX : AFBCLOAD AT 0221D8
        VABEX : AFBVLOCA AT 02E058
        VABEX : AFBVOPEP AT 42E4E0
        VABEX : AFBVSIOS AT 4222F0
        VABEX : AFBVSPIE AT 024FC0
        VABEX : AFBVSTAE AT 025118
        VABEX : AFBVTRCH AT 42D8D0
        VABEX : AFBCLCIO AT 0204F0
        VABEX : AFBVPARM AT 0247F0
        VABEX : AFBVGPRM AT 022740
        VABEX : AFBVEMGN AT 429820
        VABEX : AFBUATBL AT 025468
        VABEX : AFBUOPT  AT 025DC0
```

```
DATA SET REFERENCE NUMBER TABLE. NUMBER OF ENTRIES IS  99.

DEFAULT UNIT FOR THE OBJECT TIME ERROR MESSAGES IS   6.

DEFAULT UNIT FOR THE READER IS   5.

DEFAULT UNIT FOR THE PRINTER IS   6.

DEFAULT UNIT FOR THE PUNCH IS   7.
```

Figure 78 (Part 1 of 2). Example 3—Output for ABMODLST and ABSDUMP



```
 FILE ON UNIT   6 IS ACTIVE.
  FILE IS USED FOR SYNCHRONOUS SEQUENTIAL I/O.
  FILE STATUS IS OLD.
  FILE IS FORMATTED.
  FILE NAME USED IS FT06F001.

VPOSA - POST ABEND SYMBOLIC DUMP FOR MODULE: MAIN

 MODULE MAIN     WAS CALLED BY OP/SYS  .

 MODULE MAIN     LAST CALLED VFEIM#  .
 FROM OFFSET 000226.

I                  I4      99999999
-------------------------------------------------------------------------------

ARRAY:  A   TYPE:R4

  DIMENSION 1: (       1:      10)
  A( #)
    # = 5        0.320000E+01

 ARRAY ELEMENTS WITH A VALUE OF ZERO OR BLANK ARE NOT PRINTED.
-------------------------------------------------------------------------------

END OF SYMBOL DUMP PROCESSING FOR MAIN    .


PROGRAM UNIT NOT COMPILED FOR SYMBOLIC DUMP PROCESSING. PROGRAM UNIT IS VFEIM#

 MODULE VFEIM#   WAS CALLED BY MAIN    .
 FROM OFFSET 000226.

 MODULE VFEIM#   LAST CALLED AFBVSIOS.
 FROM OFFSET 000560.


PROGRAM UNIT NOT COMPILED FOR SYMBOLIC DUMP PROCESSING. PROGRAM UNIT IS AFBVSIOS
DMSABN155T USER ABEND 0240 CALLED FROM 02F0D8.
```

Figure 78 (Part 2 of 2). Example 3—Output for ABMODLST and ABSDUMP

Appendix D. Library Procedures and Messages

This appendix contains explanations of the program-interruption and error procedures used by the VS FORTRAN Version 2 library. The messages generated by that library are also given. A full description of program interrupts is given in *IBM System/370 Principles of Operation*, GA22-7000, and *IBM System/370 Extended Architecture Principles of Operation*, SA22-7085. For detailed information about error processing and message formats, see *VS FORTRAN Version 2 Programming Guide.*

Library Interruption Procedures

The VS FORTRAN Version 2 library processes those interrupts that are described below; all others are handled directly by the system supervisor:

1. When an interrupt occurs, indicators are set to record exponent overflow, underflow, fixed-point, floating-point, or decimal divide exceptions. These indicators can be interrogated dynamically by the subprograms described under Chapter 7, "Service Subroutines" on page 269.
2. A message is printed on the object program error unit when each interrupt occurs. The old Program Status Word (PSW) printed in the message indicates the cause of each interrupt.
3. Result registers are changed when exponent overflow or exponent underflow (codes C and D) occurs. Result registers are also set when a floating-point instruction is referenced by an assembler language execute (EX) instruction.
4. Condition codes set by floating-point addition or subtraction instructions are altered for exponent underflow (code D).
5. After the foregoing services are performed, execution of the program continues from the instruction following the one that caused the interrupt.

Library Error Procedures

During execution, the mathematical subprograms assume that the argument(s) is the correct type. However, some checking is done for erroneous arguments (for example, the wrong type, invalid characters, and the wrong length); therefore, a computation performed with an erroneous argument has an unpredictable result. However, the nature of some mathematical functions requires that the input be within a certain range. For example, the square root of a negative number is not permitted. If the argument is not within the valid range given in Figures 26 through 31, an error message is written on the object program error unit data set defined by the installation during system generation. The execution of the program is continued with the standard corrected argument value of 0.0; however, the user can specify a user exit routine for this particular error, and in that routine specify a new argument to be used to recalculate the square root. The user exit routine is part of the extended error-handling capability of the VS FORTRAN Version 2 Library. This facility provides for standard corrective action by the user. (For a full description of extended error handling, see *VS FORTRAN Version 2 Programming Guide.*)

Library Messages

The VS FORTRAN Version 2 Library generates three types of messages:

- Operator messages
- Program-interrupt messages
- Execution error messages

Operator messages are listed under "Operator Messages" on page 376. There are seven program-interrupt messages: AFB112I, AFB116I, AFB117I, AFB207I, AFB208I, AFB209I, and AFB210I, listed in "Program-Interrupt Messages" on page 377. Execution errors are listed sequentially in "Execution Error Messages" on page 379.

All library messages are numbered. Operator messages are written when a STOP n or PAUSE statement is executed. Program-interrupt messages are written when an exception to a system restriction occurs, such as dividing by 0 or generating a result too large to contain in a floating-point register. Execution error messages are written when a library function or subroutine is misused or an I/O error occurs.

Except for operator and informational messages, all library messages are followed by additional information that identifies the name of the last-executed FORTRAN program and the location of the last-executed statement in that program unit. The additional information is indicated in one of three formats, based on how the program unit was compiled:

- Program unit compiled with NOSDUMP and NOTEST:

  ```
  LAST EXECUTED FORTRAN STATEMENT IN PROGRAM name
  (OFFSET 00000000).
  ```

- Program unit compiled with TEST and NOSDUMP under Version 1, or TEST and SDUMP(SEQ) under Version 2:

  ```
  LAST EXECUTED FORTRAN STATEMENT IN PROGRAM name
  AT ISN nnnnnn (OFFSET 00000000).
  ```

- Program unit compiled with SDUMP or, for some errors, GOSTMT:

  ```
  LAST EXECUTED FORTRAN STATEMENT IN PROGRAM name
  AT ISN nnnnnn (OFFSET 00000000).
  ```

where:

`name` is the name of the failing program unit (shareable part name if compiled with RENT).

`00000000` is the hexadecimal offset from the beginning of the program to the last-executed statement.

`nnnnnn` is the compiler-generated internal statement number (ISN).

This additional information is invaluable in determining the source of the error. It should be noted, however, that, if the last-executed FORTRAN program unit called an assembler routine that invoked the VS FORTRAN Version 2 Library routine that caused the error, the source of the error may be the user-coded assembler routine.

The additional information identifying the source of the error is not produced if no FORTRAN program units are encountered in the active chain of program units that caused issuance of the error message.

If a problem recurs after you have performed the specified programmer response for the message received, see *VS FORTRAN Version 2 Diagnosis Guide.*

Operator Messages

Operator messages for PAUSE and STOP statements may be generated during load module execution as follows:

---

yy AFB001A PAUSE x.

Explanation: A PAUSE statement has been executed. The yy is an identification number assigned to the message by the operating system. The x can be:

- An unsigned 1- to 5-digit integer constant specified in the PAUSE statement
- A character constant specified in the PAUSE statement
- A zero to indicate that the PAUSE statement contained no constant

System Action: The program enters the wait state.

Operator Response: Follow the instructions given by the programmer when the program was submitted for execution; these instructions should indicate the action to be taken for any constant printed in the message text or for a PAUSE statement without a constant.

To resume execution, reply to the outstanding console message after performing the operations requested.

AFB002I STOP x.

Explanation: A STOP statement has been executed. The x can be:

- An unsigned 1- to 5-digit integer constant specified in the STOP statement
- A character constant specified in the STOP statement

System Action: The STOP statement caused the program to terminate.

Operator Response: None.

Program-Interrupt Messages

AFB112I VBALG : PROGRAM INTERRUPT - VECTOR BOUNDARY MISALIGNMENT, PSW xxxxxxxxxxxxxxxx.

Explanation: An attempt was made to use an array that was not properly aligned (not aligned on an integral boundary) as an operand of a vector instruction.

Standard Corrective Action: For vector store instructions, the contents of the source vector register will first be stored in an aligned temporary location, and scalar instructions will then be used to move the data to the target storage locations. For other vector instructions, the section of elements of the source storage array will first be moved to an aligned temporary location, and the vector instruction will then be executed, using the aligned temporary as the source storage operand. Execution then continues with the next instruction.

Programmer Response: Make sure all arrays used in vector instructions are properly aligned. INTEGER*2 arrays should be aligned on halfword boundaries; INTEGER*4, LOGICAL*4, REAL*4, and COMPLEX*8 arrays should be aligned on fullword boundaries; REAL*8 and COMPLEX*16 arrays should be aligned on doubleword boundaries.

AFB116I VUNIN : PROGRAM INTERRUPT - VECTOR UNNORMALIZED OPERAND ON DIVIDE, PSW xxxxxxxxxxxxxxxx.

Explanation: An attempt was made to perform a vector floating-point divide instruction using an operand that contained data other than a normalized floating-point number.

Standard Corrective Action: None. The program will be abnormally terminated.

Programmer Response: Make sure all floating-point operands used in vector divide instructions are properly normalized. This exception is often caused by mistakenly passing INTEGER or LOGICAL data to a vectorized subprogram instead of floating-point data.

AFB117I VUNIN : PROGRAM INTERRUPT - VECTOR UNNORMALIZED OPERAND ON MULTIPLY, PSW xxxxxxxxxxxxxxxx.

Explanation: An attempt was made to perform a vector floating-point multiply instruction using an operand that contained data other than a normalized floating-point number.

Standard Corrective Action: None. The program will be abnormally terminated.

Programmer Response: Make sure all floating-point operands used in vector multiply instructions are properly normalized. This exception is often caused by mistakenly passing INTEGER or LOGICAL data to a vectorized subprogram instead of floating-point data.

AFB207I VFNTH|VINTH : PROGRAM INTERRUPT - [VECTOR] FLOATING-POINT OVERFLOW EXCEPTION, PSW xxxxxxxxxxxxxxxx REGISTER CONTAINS nnnnnnnn.

Explanation: This message indicates that an overflow exception has occurred. This exception occurs when the magnitude of the result operation is greater than or equal to 16^{63} (approximately 7.2×10^{75}).

Supplemental Data Provided: The floating-point number (nnnnnnnn) before alteration for an exponent-overflow exception.

Standard Corrective Action: Execution continues at the point of the interrupt. For an exponent overflow in a scalar register, the result register is set to the largest possible correctly signed floating-point number that can be represented:

- Short precision (16^{63}*$(1-16^{-6})$)
- Long precision (16^{63}*$(1-16^{-14})$)
- Extended precision (16^{63}*$(1-16^{-28})$)

For an exponent overflow in a vector register, the element in the result vector register on which the exception occurred is set to the largest possible correctly-signed floating-point number.

Programmer Response: Make sure a variable or variable expression does not exceed the allowable magnitude. Verify that all variables have been initialized correctly in previous source statements and have not been inadvertently modified.

AFB208I VFNTH | VINTH : PROGRAM INTERRUPT - [VECTOR] FLOATING-POINT UNDERFLOW EXCEPTION, PSW xxxxxxxxxxxxxxxx REGISTER CONTAINS nnnnnnnn.

Explanation: The message indicates that an exponent-underflow exception has occurred. This exception occurs when the result of a floating-point arithmetic operation is less than 16^{-65} (approximately 5.4×10^{-79}).

Supplemental Data Provided: The floating-point number (nnnnnnnn) before alteration.

Standard Corrective Action: Execution continues at the point of the interrupt, with the result register or element of the result vector register set to a true zero of correct precision.

Programmer Response: Make sure that a variable or variable expression is not smaller than the allowable magnitude. Verify that all variables have been initialized correctly in previous source statements and have not been inadvertently modified.

AFB209I VFNTH | VINTH : PROGRAM INTERRUPT - yyyyyy EXCEPTION, PSW xxxxxxxxxxxxxxxx REGISTER CONTAINS nnnnnnnn.

Explanation: This message indicates that an attempt to divide by 0 has occurred.

Supplemental Data Provided: Floating-point number (nnnnnnnn) before alteration, for a floating-point interrupt. The type of interruption (yyyyy).

Standard Corrective Action: For floating-point-divide, execution continues at the point of the interrupt with the result register, or element of the result vector register, set to:

- True zero of correct precision for case of n/0, where n=0.
- Largest possible floating-point number of correct precision for case of n/0 where n≠0. For fixed-point-divide, leave registers unmodified and continue execution.

Programmer Response: Either correct the source where division by 0 is occurring, or modify previous source statements to test for the possibilities, or bypass the invalid division.

AFB210I VFNTH : PROGRAM INTERRUPT - yyyyyy EXCEPTION, PSW xxxxxxxxxxxxxxxx.

Explanation: A program interruption occurred.

Standard Corrective Action: The operation is suppressed and message AFB240I is issued.

Supplemental Data Provided: The type of interruption (yyyyyy) and the PSW at the time of the interruption (xxxxxxxxxxxxxxxx). When it appears that a vector instruction has been used by the program unit, an extra note—(VECTOR INSTRUCTION)—will be added to the message.

The type of interruption will be one of the following:

Operation exception
Privileged-operation exception
Execute exception
Protection exception
Addressing exception
Specification exception
Data exception
Fixed-point-overflow exception
Fixed-point-divide exception
Decimal-overflow exception
Decimal-divide exception
Exponent-overflow exception
Exponent-underflow exception
Significance exception
Floating-point-divide exception

The causes of these interruptions are explained in *IBM System/370 Extended Architecture Principles of Operation*, SA22-7085.

Programmer Response:

Most likely, one of the following happened:

- Your program addressed a point outside the bounds of an array and possibly wrote over program code. Make sure you refer to all arrays within the declared bounds.
- A subroutine was passed the wrong number of arguments or arguments of the wrong data type. Make sure all subroutine and function calls are passed the correct number and type of arguments.
- A call or reference was made to an external subroutine or function that has not been

resolved by the linkage editor or loader. When a program refers to an unresolved subroutine or function, an operation exception usually occurs. VS FORTRAN Version 2 indicates the location of the unresolved call or reference in the information it adds to this error message.

The PSW will probably show that the failing address is in low storage. If so, check the link-edit map and look for loader diagnostics. Make sure that external routines are available when link-editing or loading.

- A library routine caused the interruption. The information added to this message gives the name of the library routine and the offset within the routine at which the interruption occurred. If a user-coded assembler subroutine called the library routine, make sure the correct number and type of arguments were passed.
- If the note "(VECTOR INSTRUCTION)" is included on the message, then the hardware was unable to support a vector instruction. Either recompile without the VECTOR option, or run the program as compiled on a machine that supports vector processing.

Execution Error Messages

Each of these has the form:

AFBxxxI zzzzz : message text

where:

xxx is the number of the library message.

zzzzz is the last five characters of the module named AFBzzzzz.

message text describes the error.

Each message contains the error number, the abbreviated module name for the origin of the error, and a description of the error with supplemental data. In addition, a full explanation of the error is given and the standard action for correcting it is described.

Several messages have more than one format.

Variable information in the message is shown in lowercase letters. In the corrective action descriptions, • denotes the largest possible number that can be represented for a floating-point value.

AFB096I VFINP : FILEINF ARGUMENT LIST IS IN AN INCORRECT FORMAT.

AFB096I VFINP : FILEINF ARGUMENT NO. nn CONTAINS AN INVALID KEYWORD.

AFB096I VFINP : AN INCORRECT VALUE WAS GIVEN FOR THE FILEINF PARAMETER 'ppppppp'.

Explanation: The argument list for the service routine FILEINF specified incorrect data.

For the first format of the message, one of the following conditions was detected:

- The argument list had an even number of arguments
- The argument list was not in the format that is generated by the VS FORTRAN compiler when character arguments are provided.

For the second format of the message, argument nn was not one of the keyword parameters that are recognized by the FILEINF service routine.

For the third format of the message, the argument that immediately followed the keyword ppppppp contained a value not allowed for ppppppp.

Supplemental Data Provided:

nn indicates the position in the argument list of an incorrect keyword parameter. For example, in the following statement:

```
CALL FILEINF (IRETCODE, 'CYL', 50, 'S', 5)
```

the "S" is not an acceptable keyword parameter, and nn would be 4 to indicate the third argument in the argument list.

ppppppp is the keyword parameter whose corresponding value was incorrectly specified.

Standard Corrective Action: The file information provided is ignored, and an error (error number 219) will be detected during execution of a subsequent OPEN or INQUIRE statement.

Programmer Response:

For the first format of the message, assure that the argument list contains an odd number of arguments and that the even-numbered arguments are character expressions whose values are the permissible keyword parameters.

For the second format of the message, correct argument nn by coding one of the keyword parameters that is recognized by the FILEINF service routine. Be sure that the keyword parameter is coded as a character expression.

For the third format of the message, in the argument following the keyword parameter ppppppp, provide a value in the form that is allowed for the parameter ppppppp.

AFB099I DDCMP : DYNAMIC COMMON aaaaaaaa NOT AVAILABLE IN MAIN TASK PROGRAM FOR SHARING IN PARALLEL SUBROUTINE.

Explanation: A dynamic common block was not defined in the main task program before the service routine SHRCOM was called.

Supplemental Data Provided:

aaaaaaaa dynamic common block name

Standard Corrective Action: The request to share the dynamic common block is ignored and execution proceeds.

Programmer Response: Define the dynamic common within some program unit in the main task program that has been entered at least once before SHRCOM is called.

AFB100I VOPEP : OPEN STATEMENT ATTEMPTED WITH INVALID SPECIFIER FOR OPEN TO ERROR MESSAGE UNIT. UNIT nn.

Explanation: An OPEN statement for the error message unit was issued with a specifier other than the UNIT, ERR, IOSTAT or CHAR specifier.

Supplemental Data Provided:

nn the unit number of the error message

Standard Corrective Action: The OPEN statement is ignored and execution continues. If the ERR specifier was coded on the OPEN statement, control is passed to the indicated statement.

Programmer Response: Change the program to request I/O to a unit not being used for error messages or remove all specifiers that should not have been specified.

AFB101I VSIOS : MULTIPLE SUB-FILES CANNOT BE DYNAMICALLY ALLOCATED, UNIT nnn.

Explanation: An attempt was made to read or write data beyond the first sub-file which was dynamically allocated. Scratch files which do not have an explicit file definition are allocated by FORTRAN dynamically. If the scratch file is to be a multiple sub-file, each sub-file must have an explicit file definition.

Supplemental Data Provided:

nnn the unit number of the file

Standard Corrective Action: The file is closed and the READ or WRITE operation is ignored.

Programmer Response: Do not attempt to create multiple sub-files for a dynamically allocated scratch file. Supply file definitions for all sub-files including the first sub-file or change the STATUS specifier on the OPEN statement to NEW, OLD or UNKNOWN, as appropriate.

AFB102I VOPEP : A CMS FILE IDENTIFIER OR MVS DATA SET NAME IS NOT ALLOWED FOR THE FILE SPECIFIER FOR A FILE TO BE CONNECTED FOR KEYED ACCESS, FILE fffffff.

Explanation: A file to be connected for keyed access cannot specify a CMS file name, file type, file mode or a MVS data set name in the FILE specifier on the OPEN statement.

Supplemental Data Provided:

fffffff the name specified by the FILE specifier on the OPEN statement

Standard Corrective Action: The OPEN statement is ignored and execution continues.

Programmer Response: Provide a file definition statement for the ddname, and connect the file by the ddname of the file definition.

AFB103I VDYNA : ALLOCATION FAILED. NOT ENOUGH SPACE AVAILABLE ON VOLUME TO CREATE A NEW DATA SET. FILE ffffffff.

AFB103I VDYNA : ALLOCATION FAILED. REQUEST FOR EXCLUSIVE USE OF A SHARED DATA SET CANNOT BE HONORED. FILE ffffffff.

AFB103I **VDYNA : ALLOCATION FAILED. REQUESTED DATA SET NOT AVAILABLE. ALLOCATED TO ANOTHER JOB. FILE ffffffff.**

AFB103I **VDYNA : ALLOCATION FAILED. SPECIFIED VOLUME IS NOT MOUNTED. FILE ffffffff.**

AFB103I **VDYNA : ALLOCATION FAILED. INCORRECT DEVICE NAME SUPPLIED. FILE ffffffff.**

AFB103I **VDYNA : ALLOCATION FAILED. VOLUME DOES NOT HAVE ENOUGH SPACE FOR THE DIRECTORY. FILE ffffffff.**

AFB103I **VDYNA : ALLOCATION FAILED. DIRECTORY SPACE REQUESTED IS LARGER THAN THE PRIMARY. FILE ffffffff.**

AFB103I **VDYNA : ALLOCATION FAILED. REQUIRED CATALOG IS NOT AVAILABLE. FILE ffffffff.**

AFB103I **VDYNA : ALLOCATION FAILED. INSUFFICIENT SPACE IN CATALOG. FILE ffffffff.**

AFB103I **VDYNA : ssssssss FAILED. ERROR CODE xxxx, INFORMATION CODE yyyy. FILE ffffffff.**

Explanation: An OPEN or INQUIRE statement was being executed for one of the following:

- an MVS data set name specified in the FILE specifier
- a scratch file without an explicit file definition

However, an error condition was detected while running under MVS in dynamic allocation or deallocation of the data set.

For format 1 of this message, the volume allocated does not have enough space left for the creation of the new data set.

For format 2 of this message, the ACTION specifier on the OPEN statement indicated WRITE or READWRITE, so DISP=OLD was used in the allocation request. However, the same data set was already allocated by another user with DISP=SHR, so the request could not be honored. This error condition is detected if running in batch only.

For format 3 of this message, the requested data set was already allocated by another user with DISP=OLD for exclusive use, so the request could not be honored. This error condition is detected if running in batch only.

For format 4 of this message, the volume serial number specified may be invalid or does not exist.

For format 5 of this message, the device name specified may be invalid or does not exist on your system.

For format 6 of this message, there was not enough space for the directory on the volume to satisfy the request in allocating a new partitioned data set.

For format 7 of this message, space requested for the directory was greater than the primary amount specified in the service routine FILEINF call or calculated by VS FORTRAN. The space used for the directory must be less than the primary amount since the system decreases this by the amount of space allocated for the directory.

For format 8 of this message, the catalog was not available for the specified data set.

For format 9 of this message, there was not enough space available in the catalog when a new data set was being created and cataloged.

For format 10 of this message, an error occurred other than those described in the other formats for this message when allocating or deallocating a data set.

Supplemental Data Provided:

ffffffff	file name or data set name
ssssssss	ALLOCATION or DEALLOCATION
xxxx	error code
yyyy	information code

Standard Corrective Action: The I/O request is ignored and execution continues.

Programmer Response:

For format 1 of this message, specify a different volume which has more space left, or reduce the space requested, or delete unneeded data sets from the volume.

For format 2 of this message, wait until the data set has been freed or request read-only access to the data set.

For format 3 of this message, wait until the data set has been freed or allocate a different data set.

For format 4 of this message, correct the volume serial number specified.

For format 5 of this message, supply the correct device name on the call to the service routine FILEINF.

For format 6 of this message, reduce the directory space requirements or allocate the data set on another volume having more space.

For format 7 of this message, either increase the primary amount or decrease the directory size requirement.

For format 8 of this message, a JOBCAT or STEPCAT statement must be supplied in your job before you refer to the cataloged data set.

For format 9 of this message, define another catalog or remove unneeded data sets from the existing catalog.

For format 10 of this message, refer to SVC 99 return codes in *MVS/XA Programming Library: System Macros and Facilities, Volume 1*, GC28-1150, or *OS/VS2 MVS System Programming Library: Job Management*, GC28-1303. If the information code, yyyy, is 0, the test 'INFORMATION CODE yyyy' will not be printed out.

AFB104I VOPEP : AN INVALID VALUE WAS GIVEN FOR THE CHAR SPECIFIER ON THE OPEN STATEMENT, UNIT nn.

Explanation: A value other than the character expressions 'DBCS' or 'NODBCS' was specified on the CHAR specifier of the OPEN statement.

Supplemental Data Provided:

nn	unit number for which the OPEN was issued

Standard Corrective Action: The OPEN statement is ignored and processing continues. If the ERR specifier was coded on the OPEN statement, control is passed to the indicated statement.

Programmer Response: Change the CHAR specifier to CHAR='DBCS' or CHAR='NODBCS' as appropriate.

AFB105I VDIOS | VSIOS | VVIOS | CVIOS | VKIOS : FILE DELETION NOT ALLOWED FOR xxxxx, FILE ffffffff.

Explanation: STATUS='DELETE' may not be specified for close operations for certain types of files, when the OCSTATUS execution-time option is in effect or if the file has been dynamically allocated.

Supplemental Data Provided: xxxxx describes the file characteristics of file ffffffff. The file characteristics may be one of the following:

- NON-REUSABLE, NON-EMPTY VSAM FILE
- FILE OPENED WITH ACTION OF READ
- UNLABELED TAPE FILE
- CMS TAPE FILE
- SYSIN FILE
- SYSOUT FILE
- TERMINAL FILE
- UNIT RECORD INPUT FILE
- UNIT RECORD OUTPUT FILE
- SUBSYSTEM FILE
- CONCATENATED FILE
- KEYED FILE WITH AN ALTERNATE INDEX
- INPUT ONLY FILE
- FILE SPECIFIED ON THIS DEVICE
- FILE WITH MULTIPLE SUB-FILES

Standard Corrective Action: The file is disconnected, but not deleted, as if STATUS='KEEP' had been specified, and execution continues.

Programmer Response: Modify program so that file deletion is not attempted for these types of files, when the OCSTATUS execution-time option is in effect or if the file has been dynamically allocated.

AFB106I VINQP : FILE SPECIFIER WITH A BLANK VALUE IS NOT ALLOWED ON THE INQUIRE STATEMENT WITHOUT THE UNIT SPECIFIER.

Explanation: The UNIT=un specifier was omitted from the INQUIRE by unnamed file.

Standard Corrective Action: I/O request is ignored and execution proceeds.

Programmer Response: Make sure that you include a UNIT=un specifier on an INQUIRE by unnamed file.

AFB107I VOPEP : FILE NAME OF fffffff NOT ALLOWED ON THE OPEN STATEMENT.

Explanation: Unnamed files cannot be explicitly opened.

Supplemental Data Provided:

fffffff the default ddname or file name used (FTnnFmmm, FTnnKkk, FTERRsss, or FTPRTsss, or FILE FTnnFmmm, FILE FTnnKkk, FILE FTERRsss or FILE FTPRTsss)

Standard Corrective Action: I/O request is ignored and execution proceeds.

Programmer Response: Make sure that your program does not include OPEN statements for unnamed files.

AFB108I name1 : STATUS OF 'NEW' IS NOT ALLOWED ON THE OPEN STATEMENT FOR AN EXISTING FILE, FILE ffffff.

AFB108I name2 : STATUS OF 'NEW' IS NOT ALLOWED ON THE OPEN STATEMENT FOR A FILE ON A DEVICE THAT IS RESTRICTED TO INPUT ONLY, FOR FILE ffffff.

AFB108I name1 : STATUS OF 'OLD' IS NOT ALLOWED ON THE OPEN STATEMENT FOR A FILE WHICH DOES NOT EXIST, FILE ffffff.

AFB108I name1 : ACTION OF 'READ' IS NOT ALLOWED ON THE OPEN STATEMENT FOR A FILE WHICH DOES NOT EXIST, FILE ffffff.

AFB108I name2 : ACTION OF 'READ' IS NOT ALLOWED ON THE OPEN STATEMENT FOR A FILE ON A DEVICE THAT IS RESTRICTED TO OUTPUT ONLY, FILE ffffff.

AFB108I name2 : ACTION OF 'WRITE' IS NOT ALLOWED ON THE OPEN STATEMENT FOR A FILE ON A DEVICE THAT IS RESTRICTED TO INPUT ONLY, FILE ffffff.

AFB108I VSIOS : AN OUTPUT OPERATION IS BEING ATTEMPTED FOR A FILE ON A DEVICE THAT IS RESTRICTED TO INPUT ONLY, FILE ffffff.

AFB108I name3 : ACTION OF 'READ' IS NOT ALLOWED ON THE OPEN STATEMENT WITH STATUS OF 'NEW', FILE ffffff.

Explanation:

For format 1 of this message, the OCSTATUS option is in effect and you attempted to connect an existing file (or you tried to reconnect an empty file, which was previously closed during the same program) with STATUS='NEW' specified on the OPEN statement. (In the reconnection case, the empty file is considered by VS FORTRAN to still exist even after it has been closed.

For format 2 of this message, STATUS='NEW' was specified on the OPEN statement but the file is on a device that is restricted to input only. Under CMS, an attempt was made to connect a file to a read-only disk. Under MVS, an attempt was made to connect a reader, system input (SYSIN) or in-stream data set, or a file for output whose JCL specifies LABEL=(,,,IN).

For format 3 of this message, the OCSTATUS option is in effect and you attempted to connect a nonexistent file with STATUS='OLD' specified on the OPEN statement.

For format 4 of this message, the OCSTATUS option is in effect and you attempted to connect a nonexistent file with ACTION='READ' specified on the OPEN statement.

For format 5 of this message, ACTION='READ' was specified on the OPEN statement for a file that can be used for output only, such as a printer. Under MVS, an attempt was made to connect for read operations a system output (SYSOUT) data set, or a file whose JCL specifies LABEL=(,,,OUT).

For format 6 of this message, ACTION='WRITE' was specified on the OPEN statement, but the file can be used for input only. Under MVS, an attempt was made to connect for write operations a reader, SYSIN or in-stream data set, or a file whose JCL specifies LABEL=(,,,IN).

For format 7 of this message, an I/O request conflicts with how the device is accessed or connected. Under CMS, an attempt was made to write to a read-only disk. Under MVS, an attempt was made to write to a file whose JCL specifies LABEL=(,,,IN).

For format 8 of this message, ACTION='READ' was specified on the OPEN statement, but this conflicts with the specifier STATUS='NEW'. You attempted to connect a new file, that is, one that did not exist prior to connection, with the intent to read from it.

Supplemental Data Provided:

name1	VSIOS, VDIOS, or VKIOS
name2	VSIOS or VDIOS
name3	VOPEP or VSIOS
fffffff	file name

Standard Corrective Action: The I/O request is ignored and execution continues.

Programmer Response:

For format 1 of this message, make sure the STATUS specifier accurately reflects the existence property of the file you are attempting to connect. Refer to VS FORTRAN Version 2 Programming Guide for the types of files for which verification of consistency between file existence and the STATUS specifier is done.

Under CMS, if the file already exists on a minidisk, either erase it, or change the STATUS specifier on the OPEN statement to OLD. Under MVS, if the data set already exists on a volume and it is not empty, either empty it, remove it from the volume, or change the STATUS specifier on the OPEN statement to OLD.

When you want to reconnect a file after it has been closed previously during the current program, be sure to specify STATUS='OLD' whether the file is empty or not.

For format 2 of this message, either change the OPEN statement so that it does not specify STATUS='NEW', or do one of the following, depending on the system you are using: When running under CMS, find a READ/WRITE disk on which the file is to reside when it is connected. When running under MVS, change the JCL so that it does not refer to a file which can be used for input only; that is, remove the LABEL=(,,,IN) statement from the JCL, or do not use SYSIN or in-stream data sets.

For format 3 of this message, make sure the STATUS specifier accurately reflects the existence property of the file you are attempting to connect. Refer to VS FORTRAN Version 2 Programming Guide for the types of files for which verification of consistency between file existence and the STATUS specifier is done. Under CMS, make sure the file exists on a mini-disk, or change the STATUS specifier on the OPEN statement to NEW or UNKNOWN.

For format 4 of this message, make sure the STATUS specifier accurately reflects the existence property of the file you are attempting to connect. Since the file does not exist in this case, the ACTION specifier on the OPEN statement should be changed to WRITE or READWRITE.

For format 5 of this message, find a READ/WRITE disk on which to connect the file when running under CMS. When running under MVS, remove the LABEL=(,,,OUT) statement from the JCL, or do not use SYSOUT data sets.

For format 6 of this message, either change the ACTION specifier on the OPEN statement to READ, or do one of the following, depending on the system you are using: When running under CMS, find a READ/WRITE disk on which the file is to reside when it is connected. When running under MVS, remove the LABEL=(,,,OUT) statement from the JCL, or do not use SYSIN or in-stream data sets.

For format 7 of this message, when running under CMS, either change the program so that it does not perform output operations to the input-only file, or find a READ/WRITE disk on which to connect the file. When running under MVS, either change the program so that it does not perform output operations to the input-only file, or remove the LABEL=(,,,IN) statement from the JCL, or do not use the SYSIN or in-stream data sets.

For format 8 of this message, change the ACTION or the STATUS specifier to be consistent with the request. Under CMS, if the file already exists on a mini-disk, or under MVS, if the data set exists on a volume and is not empty, then change the STATUS specifier on the OPEN statement to OLD. Otherwise, change the ACTION specifier to WRITE or READWRITE.

AFB109I VINQP : A FILE NAME IS NOT ALLOWED ON THE INQUIRE STATEMENT WITH THE UNIT SPECIFIER, FILE=ffffffff.

Explanation: An INQUIRE statement with both a FILE specifier with a non-blank value and a UNIT specifier is given.

Supplemental Data Provided: ffffffff is the file name given on the FILE specifier.

Standard Corrective Action: I/O request is ignored and execution proceeds.

Programmer Response: Modify your program to use one of the three valid forms of the INQUIRE statement, that is, INQUIRE by unit, file, or unnamed file.

AFB110I VSIOS : THE I/O STATEMENT REFERS TO A UNIT THAT IS NOT CONNECTED, UNIT nn.

Explanation: A READ, WRITE, BACKSPACE, REWIND, or ENDFILE statement is given for a disconnected file.

Supplemental Data Provided: nn is the unit number.

Standard Corrective Action: I/O request is ignored and execution proceeds.

Programmer Response: Use an INQUIRE statement to determine the file connection status prior to reading from or writing to a file or use an OPEN statement to reconnect the file.

AFB111I name : FILE DELETION FAILED. SYSTEM COMPLETION CODE ccc-rr. FILE ffffffff.
AFB111I name : FILE DELETION FAILED. READ-ONLY DISK. FILE ffffffff.
AFB111I name : FILE DELETION FAILED. UNEXPECTED ERASE RETURN CODE rc. FILE ffffffff.
AFB111I name : FILE DELETION FAILED. VSAM OPEN MACRO RETURN CODE vrc, ERROR CODE X'hc' (dc).

Explanation: The file could not be deleted for the reason specified. One of the following caused the file deletion to be attempted:

- A CLOSE statement with STATUS='DELETE' was specified.
- A CLOSE statement was issued for a file that was connected with STATUS='SCRATCH'.
- Program termination caused an implicit CLOSE operation for a file that was connected with STATUS='SCRATCH'.

Supplemental Data Provided:

name	VSIOS, VDIOS, CVIOS, VVIOS, or VKIOS
ffffffff	file name
rr	reason code
rc	CMS ERASE error return code
vrc	VSAM return code
hc	VSAM error feedback code in hexadecimal
dc	VSAM error feedback code in decimal

Standard Corrective Action: The file is disconnected, but not deleted, as if STATUS='KEEP' had been specified, and execution proceeds.

Programmer Response: Modify program by removing the STATUS='DELETE' specifier from the CLOSE statement, or by connecting the file with a status of NEW, OLD, or UNKNOWN.

For format 1 of the message (MVS only), the file may have been RACF protected. Check the system completion code. For more information on system completion codes, see *MVS/370 Message Library System Codes*, GC38-1008, or *MVS/XA Message Library: System Codes*, GC28-1157.

For format 2 of the message (CMS only), issue the CP LINK command to reset the disk linkage to read/write mode, then reaccess the disk.

For format 3 of the message (CMS only), information on on the ERASE command return codes can be found in *VM/SP CMS Command and Macro Reference*, SC19-6209, or *VM/SP System Messages and Codes*, SC19-6204.

For format 4 of the message, an attempt was made to delete a VSAM file. For an explanation of VSAM codes, see *MVS/XA VSAM Administration: Macro Instruction Reference*, GC26-4016 (for DFP Version 1) or GC26-4152 (for DFP Version 2).

AFB112I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 377.

AFB113I VIADI : THE RECONNECT COMMAND REFERS TO A UNIT WHICH CANNOT BE RECONNECTED TO FILE ffffffff.

Explanation: The IAD RECONNECT command was issued to reconnect a unit to a VSAM or non-sequential file.

Supplemental Data Provided:

ffffffff The file name

Standard Corrective Action: The RECONNECT command is ignored and execution continues.

Programmer Response: Change the operating system file definition statement refers to a VSAM or non-sequential file.

AFB114I VDIOS : A FILE BEING CONNECTED FOR DIRECT ACCESS CAN RESIDE ONLY ON DASD, FILE ffffffff.

AFB114I VDIOS : A FILE BEING CONNECTED FOR DIRECT ACCESS CANNOT BE IN A PDS MEMBER, FILE ffffffff.

Explanation: The FILEDEF command, ALLOCATE command, or DD statement used for direct access I/O was determined to be connected to an unusable device type. The only acceptable device type is DASD, non-PDS. Files such as terminal, reader, SYSIN, and SYSOUT are not acceptable.

Supplemental Data Provided:

ffffffff file name

Standard Corrective Action: The OPEN request is ignored and execution continues. If the ERR specifier was coded on the OPEN statement, execution begins at the statement indicated on the ERR specifier.

Programmer Response: Change the system file definition statement to point to a DASD file.

AFB115I CFIST : PROGRAM INTERRUPT WHILE USING VSAM. VSAM NOT ACTIVE. FILE ffffffff.

Explanation: There was a DLBL command in effect for the file **ffffffff**, but a program interrupt occurred during an attempt to refer to the file. The probable cause of this error is that VSAM is not active because no DLBL command was issued to refer to the VSAM master catalog.

Supplemental Data Provided:

ffffffff file name

Standard Corrective Action:

The I/O request is ignored and processing continues.

Programmer Response: Make VSAM active by accessing the disk that contains the VSAM master catalog. Then provide a DLBL command with a ddname of IJSYSCT to refer to the VSAM master catalog.

AFB116I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 377.

AFB117I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 377.

AFB118I FRXPR : REAL*4 BASE=base LESS THAN ZERO, REAL*4 EXPONENT=exponent

Explanation: For an exponentiation operation (R**S) in the subprogram AFBFRXPR (FRXPR#), where R and S represent REAL*4 variables or REAL*4 constants, R is less than zero.

Standard Corrective Action: Result = |base|**exponent.

Programmer Response: Make sure that both the real variable or constant base and exponent for an exponentiation operation are within the allowable range during program execution, then either modify the operands or insert source code to test for the situation and make appropriate compensation. Bypass the exponentiation operation if necessary.

AFB119I FDXPD : REAL*8 BASE=base LESS THAN ZERO, REAL*8 EXPONENT=exponent

Explanation: For an exponentiation operation (D**P) in the subprogram AFBFDXPD (FDXPD#), where D and P represent REAL*8 variables or REAL*8 constants, D is less than zero.

Standard Corrective Action: Result = |base|**exponent.

Programmer Response: Make sure that both the real variable or constant base and exponent for an exponentiation operation are within the allowable range during program execution, then either modify the operands or insert source code to test for the situation and make appropriate compensation. Bypass the exponentiation operation if necessary.

AFB120I VOPEP : OPEN STATEMENT ATTEMPTED TO CHANGE pppppppp FOR FILE fffffff WHICH IS ALREADY OPEN. ONLY 'BLANK' OR 'CHAR' MAY BE CHANGED.

Explanation: An OPEN statement was issued for a file that is already connected. The OPEN statement contains a specifier whose value has already been set and cannot be changed. When a file is already connected, only the BLANK and CHAR specifier can be specified on an OPEN statement.

Supplemental Data Provided:

pppppppp the specifier on the OPEN statement whose value cannot change. It can be ACCESS, FORM, ACTION, KEYS, STATUS, RECL or PASSWORD.

fffffff the name of the file that is already connected.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: If you want to change the value of the BLANK or CHAR specifier, first remove the specifiers that should not have been specified. Otherwise, remove the OPEN statement or connect a different file with it.

AFB121I VKIOS : OPEN STATEMENT FOR FILE fffffff SPECIFIES ACTION='WRITE' BUT HAS MORE THAN ONE KEY IN 'KEYS' SPECIFIER.

Explanation: An OPEN statement has conflicting specifiers: ACTION='WRITE', which implies you are loading a file, and KEYS with more than one key listed.

Supplemental Data Provided: fffffff is the name of the file you tried to open.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: If you want to load the file, remove the KEYS specifier or specify only the primary key of the file. If you want to process a file that is not empty, change the value of the ACTION specifier to READ or READWRITE.

AFB122I VSIOS | VDIOS | VVIOS | CVIOS | VKIOS : ssssssss STATEMENT IS NOT ALLOWED WHEN THE FILE IS OPEN WITH AN ACTION OF 'dddddddd'. FILE fffffff.

Explanation: The value of the ACTION specifier on an OPEN statement conflicts with a statement that follows the OPEN statement for the connected file. For an ACTION specifier with the value of WRITE, the READ statement is not allowed. For an ACTION specifier with the value of READ, the WRITE statement is not allowed.

In addition, for files connected for keyed access, for an ACTION specifier with the value of WRITE, the REWRITE, DELETE, REWIND, or BACKSPACE statement is not allowed. For an ACTION specifier with the value of READ, the REWRITE or DELETE statement is not allowed.

Supplemental Data Provided:

ssssssss is the name of the incompatible statement.

dddddddd is the value of the ACTION specifier that is in use.

fffffff is the name of the file.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the ACTION specifier or remove the incompatible statement.

AFB123I VKIOS : ssssssss STATEMENT IS NOT ALLOWED FOLLOWING ssssssss STATEMENT WHICH RESULTED IN cccccccccccccccc CONDITION. FILE fffffff.

Explanation: A statement was not allowed because a previous statement caused an error and the loss of position in the file being processed. You cannot read records sequentially or use a BACKSPACE, DELETE, or REWRITE statement until you have reestablished file position.

Supplemental Data Provided:

ssssssss (first occurrence) the name of the statement that was not allowed

ssssssss (second occurrence) the name of the earlier statement that caused the error

cccccccccccccccc RECORD NOT FOUND, DUPLICATE ERROR, END OF FILE, VSAM I/O ERROR, or PROGRAM LOGIC ERROR

fffffff the name of the file

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Code either a REWIND or a direct-access READ statement after the statement that caused the error. This will reestablish a position in the file and enable other input/output statements to be processed.

AFB124I VKIOS : KEYID SPECIFIER ON THE READ STATEMENT CONFLICTS WITH THE NUMBER OF KEYS IN THE KEY SPECIFIER ON OPEN STATEMENT FOR FILE fffffff.

Explanation: The value of the KEYID specifier is larger than the number of start-end pairs in the KEYS specifier. Therefore, no pair (and hence no key) can be associated with the KEYID specifier. This conflict can arise even if no KEYS specifier is coded: a default of one key is assumed, so if KEYID has a value greater than 1, an error exists.

Supplemental Data Provided:

fffffff is the name of the file for which the READ statement was issued.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the KEYID specifier so that it is no larger than the number of start-end pairs in the KEYS specifier, or remove the KEYID specifier.

AFB125I VKIOS : KEY ARGUMENT ON READ STATEMENT HAS A LENGTH OF nnnnnnnn WHICH IS GREATER THAN THE KEY LENGTH OF mmmmmmmm [(KEYID IS k).] FILE fffffff.

Explanation: The argument to be used in searching for a key was given in the KEY, KEYGE, or KEYGT specifier of a READ statement. This argument is longer than the key being searched for.

Supplemental Data Provided:

nnnnnnnn is the length in bytes of the search (or key) argument.

k is the relative position in a list of keys of the key of reference—the key currently in use. The list of keys is in the KEYS specifier of the OPEN statement. ("KEYID IS k" is omitted if the KEYS specifier of the OPEN statement specifies only one key or was not coded.)

mmmmmmmm is the length in bytes of the key being used.

fffffff is the name of the file for which the READ statement was issued.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Specify a search argument in the KEY, KEYGE, or KEYGT specifier whose length does not exceed that of the key you are searching for. If you want to search with a different key of reference, specify a different value for the KEYID specifier.

AFB126I VKIOS : RECORD NOT FOUND WITH SPECIFIED KEY. FILE fffffff. [KEYID IS k: sssss:eeeee.] xxxxx SPECIFIER VALUE IS vvvvvvvvvv.

Explanation: There was no record in the file meeting the search argument in the KEY, KEYGE, or KEYGT specifier of the READ statement. The search was based on the key specified in the KEYID specifier of the READ statement. (If there was no KEYID specifier in the READ statement, the search was based on the KEYID specifier last used. If no KEYID specifier has been used since the file was opened, the first key specified in the KEYS specifier of the OPEN statement was used for the search.)

Supplemental Data Provided:

fffffff is the name of the file for which the READ statement was issued.

k is the relative position in a list of keys of the key of reference—the key currently in use. The list of keys is in the KEYS specifier of the OPEN statement. (This part of the message and the sssss:eeeee information are omitted if the KEYS specifier of the OPEN statement specified only one key or was not coded.)

sssss is the starting position in each record of the key being used, and eeeee is the ending position.

xxxxx is KEY, KEYGE, or KEYGT—whichever specifier was used in the READ statement.

vvvvvvvvvv is the value of the specifier.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the KEY, KEYGE, or KEYGT specifier so that the appropriate record will be found. If you want to allow for the possibility of a "record not found" condition, add a NOTFOUND specifier to your program. It specifies the statement to be given control when this condition occurs.

AFB127I VIOUF | VIOFM : THE ssssssss STATEMENT REFERS TO UNIT nn WHICH IS NOT CONNECTED.

Explanation: An input/output statement referred to a unit that was not opened with an OPEN statement.

Supplemental Data Provided:

ssssssss is the name of the input/output statement—for example, READ, REWRITE, DELETE.

nn is the unit number referred to in the input/output statement.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the program to issue an OPEN statement with the ACCESS='KEYED' specifier before issuing the input/output statement.

AFB128I VKIOS : THE ssssssss STATEMENT REFERS TO FILE fffffff WHICH IS NOT A VSAM KSDS.

Explanation: An input/output statement was issued that can apply only to a VSAM file. The file, however, was opened as a non-VSAM file.

Supplemental Data Provided:

ssssssss is the name of the input/output statement.

fffffff is the name of the file.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: If you want to access a VSAM file, change the operating system's data definition statement to specify a VSAM file.

AFB129I VKIOS : THE KEYED FILE RECORD SUPPLIED BY THE ssssssss STATEMENT HAD A LENGTH OF 0. FILE fffffff.

AFB129I VKIOS : THE KEYED FILE RECORD SUPPLIED BY THE ssssssss STATEMENT HAD A LENGTH OF nnnnn WHICH IS TOO SHORT. FILE fffffff.

Explanation: Either a WRITE or REWRITE statement built a record that was too short to contain all the keys that are available (as specified by the KEYS specifier of the OPEN statement or implied by the operating system's data definition statement).

Supplemental Data Provided:

ssssssss is either WRITE or REWRITE.

nnnnn is the length of the record that was built.

fffffff is the name of the file involved in the input/output operation.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the output list of the WRITE or REWRITE statement so that it builds a record that is long enough to include all the keys.

AFB130I VKIOS : ERROR ON VSAM FILE WHILE PROCESSING ssssssss STATEMENT FOR FILE fffffff. VSAM mmmmm MACRO, RETURN CODE rc, ERROR CODE X'hc' (dc), FUNCTION CODE fc.

Explanation: VSAM detected an error while processing an input/output statement.

Supplemental Data Provided:

ssssssss is the name of the statement being processed.

fffffff is the name of the file involved in the input/output operation.

mmmmm is the name of the VSAM macro that was issued (GET, PUT, POINT, and so on).

rc is the VSAM return code.

hc is the VSAM error feedback code in hexadecimal.

dc is the same code in decimal.

fc is the function code in hexadecimal.

You can find an explanation of the codes in *OS/VS Virtual Storage Access Method (VSAM) Programmer's Guide*, GC26-3838.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Take the action given in the appropriate manual.

AFB131I VKIOS : CONFLICTING DDNAMES WOULD BE REQUIRED FOR FILE ffffffff SINCE THERE ARE k KEYS LISTED IN THE KEYS SPECIFIER ON THE OPEN STATEMENT.

Explanation: When opening files for multiple-key processing, VS FORTRAN Version 2 generates unique names for the files not named explicitly in the OPEN statement. It does this by appending a number (beginning with 1) to the end of the file name specified in the OPEN statement. If this file name has a maximum length of 7-characters, a number cannot be appended, so the last character is overlaid by a number. An error occurred in this case because the file name is 7-characters long and *ends in a number that is smaller than the number of keys specified in the OPEN statement*. If VS FORTRAN Version 2 proceeded to generate file names, it would duplicate the file name given in the OPEN statement.

Supplemental Data Provided:

fffffff is the name of the file.

k is the number of key specified in the KEYS specifier of the OPEN statement.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the file name in the OPEN statement to one that has:

- Fewer than 7 characters, or
- An alphabetic character in the last position, or
- A number in the last position that is not less than k.

AFB132I VKIOS : FILE ffffff HAS A RECORD LENGTH OF r, BUT RELATED FILE f2 HAS A DIFFERENT LENGTH OF r2.

Explanation: In attempting to open VSAM files for multiple-key processing, VS FORTRAN Version 2 found that the files had different maximum record lengths. Therefore, the data definition statements for the files must contain an error or inconsistency. For example, a statement may refer to an alternate-index file rather than to a *path* from the alternate-index file to the base cluster. Or statements may point to alternate-index files for different base clusters. Or they may mistakenly refer to two base clusters and no alternate-index files.

Supplemental Data Provided:

ffffff is the file name.

f2 is the related file.

r is the record length of the file.

r2 is the record length of the related file.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the data definition statements (DD statements in OS/VS, DLBL statements in VM) to refer to the VSAM files that represent the same base cluster.

AFB133I VKIOS : MORE THAN ONE KEY SPECIFIED IN OPEN STATEMENT FOR VSAM KSDS, BUT FILE ffffff IS EMPTY AND CANNOT BE PROCESSED.

Explanation: While opening VSAM files for multiple-key processing, VS FORTRAN Version 2 found that one of the files was empty.

Supplemental Data Provided: ffffff is the ddname of the empty file.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Be sure that the correct VSAM files are specified in the operating system's data definition statements. Also, be sure that the base cluster (the file with the primary key) has been loaded and that the other files (those with alternate-index keys) have had their alternate indexes built successfully using the Access Method Services BLDINDEX command.

AFB134I VKIOS : OPEN STATEMENT FOR THE KEYED FILE fffffff SPECIFIES A KEY OF sssss:eeeee, BUT NONE OF THE DDNAMES FOR THIS FILE CORRESPOND TO A VSAM FILE WITH THIS KEY.

Explanation: A key specified on the OPEN statement does not correspond to any of the files, specified by ddnames, that were opened for keyed access.

Supplemental Data Provided:

fffffff is the name of the file, specified explicitly or taken by default, in the OPEN statement.

sssss is the starting position in each record of the key to be used; eeeee is the ending position.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Correct the starting and ending positions of the keys in the KEYS specifier; each key must correspond to a file that is identified in a data definition statement (a DD statement in OS/VS, a DLBL statement in VM). (The keys need not be listed in the order of the data definition statements, however.) In calculating the starting and ending positions, remember that the first position in a record is position 1. This differs from the way the starting position of a key is calculated in the KEYS specifier of the Access Method Services DEFINE command. There, the first position in a record is position 0.

AFB135I VKIOS : ATTEMPT MADE TO ADD A RECORD WITH A DUPLICATE KEY TO A KEYED FILE. FILE fffffff. THE KEY OF REFERENCE HAS A KEYID OF k, A POSITION OF sssss:eeeee, AND A VALUE OF vvvvvvvvvv (HEX).

Explanation: A keyed file was opened with an ACTION value of READWRITE, and a WRITE operation tried to add a record with a duplicate key. The key duplicates either a primary key or an alternate-index key that does not allow duplicate keys. The duplicate key is *not necessarily* the key of reference, the key currently in use and described in the message. The duplicate key may not even be among the keys listed in the KEYS specifier of the OPEN statement for the file.

Supplemental Data Provided:

fffffff is the name of the file.

k indicates the key of reference—that is, the start-end pair in the KEYS specifier of the file's OPEN statement that was used in writing the record..

sssss:eeeee is the position in the record of the key of reference.

vvvvvvvvvv is the value of the key of reference.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the item in the I/O list that represents the key to be written. If you want to allow for a "duplicate key" condition in your program, code a DUPKEY specifier. It identifies the statement to be given control if the condition occurs.

AFB136I VOPEP : AN INVALID VALUE WAS GIVEN FOR THE ACTION SPECIFIER ON THE OPEN STATEMENT. UNIT nn.

Explanation: The ACTION specifier on the OPEN statement specified a value other than READ, WRITE, or READWRITE.

Supplemental Data Provided: nn is the unit number specified in the OPEN statement.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the ACTION specifier to READ, WRITE, or READWRITE.

AFB137I VOPEP : KEYS SPECIFIER ON AN OPEN STATEMENT IS NOT ALLOWED EXCEPT FOR KEYED ACCESS. UNIT nn.

Explanation: The OPEN statement has a KEYS specifier, but has either no ACCESS specifier or one whose value is incompatible with KEYS. (Only the value KEYED is compatible.)

Supplemental Data Provided: nn is the unit number specified on the OPEN statement.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: If the file to be open has keys, specify ACCESS='KEYED'. Otherwise, remove the KEYS specifier from the OPEN statement.

AFB138I VKIOS : ATTEMPT WAS MADE TO CONNECT AN EMPTY KEYED FILE USING A VALUE OF READ FOR THE ACTION SPECIFIER. VSAM ERROR CODE X'6E' (110). FILE fffffff.

Explanation: VSAM does not allow an empty file to be opened for input operations.

Supplemental Data Provided: ffffff is the name of the file for which the OPEN statement was issued.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Be sure that the correct VSAM file was specified in the operating system's data definition statement. If the file is a base cluster (the file with the primary key), be sure that it was loaded. If the file is a path for an alternate index, be sure the alternate index was built successfully using the Access Method Services BLDINDEX command.

If you want to process the base cluster and open it for retrieval operations, use ACTION='READWRITE'. This causes a dummy statement to be loaded and deleted, and VSAM then does not consider the file to be empty.

AFB139I VKIOS : ATTEMPT MADE TO REWRITE A RECORD IN WHICH THE VALUE OF THE KEY OF REFERENCE DIFFERS FROM THE VALUE OF THAT KEY IN THE RECORD JUST READ. THE KEY OF REFERENCE HAS A KEYID OF k. FILE fffffff.

Explanation: You read a record and, in trying to rewrite it, wrote a key of reference whose value differed from that in the original record.

Supplemental Data Provided: The name of the file (fffffff) and, if the file has multiple keys, the KEYID (k) of the key of reference.

Standard Corrective Action: Execution continues, but the I/O request is ignored.

Programmer Response: If you did not intend to write a new key value, make sure that:

- The I/O list contains all the fields of the record to be rewritten, and

- Changes in the order or length of non-key fields have not caused the position of the key of reference to change.

If, however, you intended to replace the record with one having a new key value, delete the record and then add a new record with the WRITE statement.

AFB140I VKIOS : KEY SEQUENCE ERROR LOADING A KEYED FILE. FILE fffffff. THE KEY OF REFERENCE IN THE REJECTED RECORD HAD A VALUE OF vvvvvvvvvv.

Explanation: You attempted to load a record in which the value of the primary key was not greater than the value of the primary key in the previous record.

Supplemental Data Provided: The name of the file (fffffff) and the value of the key of reference in the record that could not be written (vvvvvvvvvv).

Standard Corrective Action: Execution continues, but the record has not been written.

Programmer Response: Change the logic of your program or the order of the records being loaded so that the records are loaded in increasing sequence of their primary key values. Be sure that the key of reference is actually the file's primary key.

AFB141I VCOM2 : RESIDENCY ABOVE 16 MB NOT POSSIBLE RUNNING IN LINK MODE.

Explanation: You are running your program in link mode, and your program resides at an address greater than 16 Mb in an MVS/XA system. Execution is impossible in this case, because several library routines must run at an address below 16 Mb.

Supplemental Data Provided: None.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Either:

1. Do not supply the library SYS1.VLNKMLIB (or the equivalent at your installation) in the SYSLIB DD statement in the linkage editor step when link-editing your program for execution in load mode, or
2. When executing in link-mode, be sure that your load module is given an RMODE value of 24 when it is link-edited. You probably specified an RMODE of ANY; either remove this linkage editor specifier or specify an RMODE value of 24.

AFB142I VCOM2 : AFBVLBCM IS AT LEVEL lbcm-lvl BUT mod-name IS AT LEVEL mod-lvl.

Explanation: You were running your program in load mode, which requires loading the composite module *mod-name*. However, the module loaded was from a different release level of the Library than the rest of the executing library routines (in particular, different from the release level of the composite module AFBVLBCM). If you are running under CMS, the composite module *mod-name* may be in a discontiguous shared segment.

Supplemental Data Provided: The name of the loaded composite module (*mod-name*), its level (*mod-lvl*), and the level of the executing version of AFBVLBCM (*lbcm-lvl*). The levels are in the form vvrrmm, where vv is the version number, rr is the release number, and mm is the modification number.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Be sure that you are specifying for execution the correct libraries that contain the VS FORTRAN Version 2 Library. Specify:

- A JOBLIB or STEPLIB DD statement in MVS, or
- A GLOBAL LOADLIB command in CMS.

In addition, be sure that any shareable copies of the composite module are at the same level as the rest of the Library you are using for execution. These shareable copies are in:

- A link pack area in MVS
- A discontiguous shared segment in CMS

For further assistance, refer the problem to the people at your installation who give system support for VS FORTRAN Version 2.

AFB143I VDIVP : DATA-IN-VIRTUAL SERVICE dddddd FAILED FOR DYNAMIC COMMON aaaaaaaa. ABEND CODE ccc, REASON CODE rr.

AFB143I VDIVP : DATA-IN-VIRTUAL SERVICE dddddd FAILED FOR DYNAMIC COMMON aaaaaaaa. COMPLETION CODE ccc, REASON CODE rr.

AFB143I VDIVP : DATA-IN-VIRTUAL SERVICE dddddd FAILED FOR LACK OF HARDWARE SUPPORT.

Explanation:

For format 1 of this message, a DIV utility subroutine (dddddd) was invoked and a failure was detected by the system that resulted in a system abend.

For format 2 of this message, a DIV utility subroutine (dddddd) was invoked and a failure was detected by the library data-in-virtual processing routine.

For format 3 of this message, a DIV utility subroutine (dddddd) was invoked and a failure was detected by the system that resulted in a program exception that indicated that the required machine instructions were not available.

Supplemental Data Provided:

dddddd DIV macro name (UNIDENTIFY, IDENTIFY, MAP, SAVE, RESET, UNMAP, or ACCESS)

ccc abend code (for format 1) or completion code/return code (for format 2)

rr reason code

aaaaaaaa dynamic common block name or (NONE), which indicates that no dynamic common was associated when the DIV service was being processed.

Standard Corrective Action: Execution of the DIV service terminates with a return code of 128, when the failure is not interpretable. Data in a traceback listing may be inaccurate depending upon how far along system services were when the abend occurred.

Programmer Response: Take the appropriate action based on the explanation of the abend or return codes found in *MVS/XA Supervisor Services and Macro Instructions*, GC28-1154, or in *MVS/XA Message Library: System Codes*, GC28-1157.

AFB144I VDIVP : DATA-IN-VIRTUAL SERVICES ARE NOT AVAILABLE ON THIS SYSTEM.

AFB144I VDIVP : DATA-IN-VIRTUAL SERVICES ARE NOT ALLOWED IN A PARALLEL SUBROUTINE.

Explanation:

For format 1 of this message, a DIV utility subroutine was invoked on a system that does not support Data-in-Virtual (that is, on VM or on non-XA MVS).

For format 2 of this message, a DIV utility subroutine was invoked by a parallel subroutine (using MTF).

Supplemental Data Provided: None.

Standard Corrective Action: Processing resumes and the service call is ignored.

Programmer Response:

For format 1 of this message, remove the DIV utility subroutine calls or run the program on an MVS /XA system which has Data-in-Virtual support.

For format 2 of this message, remove the Data-in-Virtual utility subroutine calls from any parallel subroutines if using the multitasking facility (MTF). All of the DIV subroutine calls must be made from the main task program.

AFB145I VCOM2 : COMPOSITE MODULE mod-name IS NOT IN THE EXPECTED FORMAT.

Explanation: You were running your program in load mode, which requires loading the composite module *mod-name*. However, the module loaded was not recognized as a valid composite module. If you are running under CMS, this composite module may be in a discontiguous shared segment that was not built properly.

Supplemental Data Provided: The name of the composite module (*mod-name*.)

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Be sure you are specifying for execution the correct libraries containing the VS FORTRAN Version 2 Library with:

- A JOBLIB or STEPLIB DD statement in MVS, or
- A GLOBAL LOADLIB command in CMS.

Be sure that the composite module has been built properly. Building composite modules is explained in *VS FORTRAN Version 2 Installation and Customization for VM* and *VS FORTRAN Version 2 Installation and Customization for MVS*.

If you are executing under CMS and the system name of a discontiguous shared segment has been defined, be sure the shared segment has been built properly.

For further assistance, refer the problem to the people at your installation who give system support for VS FORTRAN Version 2.

AFB146I VLINP : THE SHAREABLE LOAD MODULE module-name WAS LOADED ABOVE THE 16MB VIRTUAL STORAGE LINE BY THE NONSHAREABLE PART OF PROGRAM program-name, WHICH WAS RUNNING IN 24-BIT ADDRESSING MODE.

Explanation: The shareable load module contains the program's shareable part, but, because of the module's location and the program's addressing mode, the program can never branch to that part. An abend would occur if it tried to branch.

Supplemental Data Provided: The names of the load module and the program.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Either·

- Run the program in 31-bit addressing mode by link-editing it with an AMODE value of 31, or
- Link-edit the shareable load module with an AMODE value of 24.

AFB147I VLINP : THE SHAREABLE LOAD MODULE module-name LOADED BY THE NONSHAREABLE PART OF PROGRAM program-name HAS AN INCORRECT FORMAT.

Explanation: A program's nonshareable part loaded a load module containing the program's shareable part. The load module, however, was not in the correct format, because the parts were not correctly separated after the program was compiled.

Supplemental Data Provided: The names of the load module and the program.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Use the object-deck separation tool to separate the shareable and nonshareable parts of the program. Then link-edit the shareable part to create the load module.

AFB148I VLINP : THE SHAREABLE LOAD MODULE module-name LOADED BY THE NONSHAREABLE PART OF PROGRAM program-name DOES NOT CONTAIN THE SHAREABLE PART shareable-part-name AT AN ACCESSIBLE LOCATION.

Explanation: A program's nonshareable part loaded a load module that does not contain the program's shareable part.

Supplemental Data Provided: The names of the load module, the program, and the parts.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Link-edit the shareable part (produced by the object-deck separation tool) into the load module.

AFB149I VLINP : THE SHAREABLE LOAD MODULE module-name LOADED BY THE NONSHAREABLE PART OF PROGRAM program-name HAS A TIMESTAMP IN THE SHAREABLE PART shareable-part-name WHICH DIFFERS FROM THAT IN NONSHAREABLE PART nonshareable-part-name.

THE shareable-part-name TIMESTAMP IS xxxxxxxxxxxxxx, AND THE shareable-part name TIMESTAMP IS yyyyyyyyyyyyyy.

Explanation: A program's nonshareable part loaded a load module containing the program's shareable part, but the timestamps of the parts were found to be different. The parts were therefore compiled at different times and are assumed to be incompatible.

Supplemental Data Provided: The names of the load module, the program, and the parts, and the timestamps (xxxxxxxxxxxxxx, yyyyyyyyyyyyyy) of the parts.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Make the load module containing the shareable part available at execution time. Tell your system programmer that the shared segment (in VM) or the link pack area (in MVS) may have to be updated.

AFB150I VLOAD : THE LOAD LIST, WHICH HAS nn ENTRIES, IS FULL. A TOTAL OF mm LOADED MODULES WAS NOT ENTERED INTO THE LIST. SUCH MODULES ARE NOT DELETED.

Explanation: While one module was loading another, the load list was found to be full. Consequently, the name and address of the loaded module cannot be added to the list.

Supplemental Data Provided: The number of entries (nn) in the load list and the number of modules (mm) whose names could not be entered into the load list.

Standard Corrective Action: Execution continues normally, but the loaded module will not be deleted when the program terminates.

Programmer Response: None.

AFB151I VDIOS : nnnn RECORDS OF LENGTH llll FORMATTED ON FILE fffffff.

Explanation: The message tells how many records were formatted on a file and how long the records are. This action was taken in response to an OPEN statement in a program accessing a new direct-access file for the first time.

Supplemental Data Provided:

nnnn is the number of records formatted on the file.

llll is the length of the records.

fffffff is the name of the file.

Standard Corrective Action: None.

Programmer Response: None.

AFB152I VSIOS | VDIOS : FILE IS UNUSABLE, PERMANENT ERROR HAS BEEN DETECTED. FILE fffffff.

Explanation: An attempted I/O operation on a file resulted in a permanent I/O error. The message that precedes this one describes the error.

Supplemental Data Provided: fffffff is the name of the unusable file.

Standard Corrective Action: The interrupted instruction and the I/O request are ignored. After the traceback is completed, control is returned to the call routine statement designated in the ERR specifier of an I/O statement, if that specifier was specified. Also, the IOSTAT variable is set to 152 if IOSTAT was specified in the I/O statement.

Programmer Response: Check the previous error message and correct the situation.

AFB153I VCOM2 : THE SPECIFIER LIST RECEIVED FROM rrrrrr IS INCONSISTENT WITH THE ARGUMENTS EXPECTED BY ssssss. INHERITED LENGTH OF A CHARACTER ARGUMENT IS REQUIRED. EXECUTION IS TERMINATED.

Explanation: A dummy argument within a subprogram (ssssss) is of character type with an inherited length; that is, it is defined as CHARACTER*(*). Such a dummy argument requires a secondary argument list that contains the lengths of the character data passed to the routine. However, this secondary argument list was not available. This situation could occur either because the calling program (rrrrrr) was compiled with a FORTRAN compiler whose level is earlier than VS FORTRAN Version 1 Release 3, or because the calling program was not a FORTRAN program. In either case, the routine was compiled with VS FORTRAN Version 1

Release 3 or later or with VS FORTRAN Version 2.

Supplemental Data Provided: The name of the calling program (rrrrrr) and the name of the called routine (ssssss).

Programmer Response:

1. If the calling program is a FORTRAN program at the FORTRAN 77 language level, then recompile it with VS FORTRAN Version 1 Release 3 or later or with VS FORTRAN Version 2.
2. If the calling program is a FORTRAN program at the FORTRAN 66 language level, then it does not generate the secondary argument list with the character lengths. To correct this situation, either convert the calling program to the FORTRAN 77 language level and compile it with VS FORTRAN Version 1 Release 3 or later or with VS FORTRAN Version 2, or change the routine so the character data in the dummy argument list is known, rather than of inherited length.
3. If the calling program is written in an Assembler language, provide both a secondary argument list with the character lengths, and the identifier that precedes the argument list. For the expected format for this information, refer to the section on character argument linkage conventions in *VS FORTRAN Version 2 Programming Guide.*
4. If the calling program is neither a FORTRAN nor an Assembler language program, the secondary argument list cannot be generated. In this case, change the routine so the character data in the dummy argument list is of known, rather than of inherited, length.

AFB154I VDIVP : BAD ARGUMENT LIST PASSED TO DATA-IN-VIRTUAL PROCESSING ROUTINE.

Explanation: A Data-in-Virtual utility subroutine was invoked with an argument list that could not be used.

Supplemental Data Provided: None.

Standard Corrective Action: Processing resumes, the service call is ignored, and, if possible, a return code is passed to the caller in the first argument.

Programmer Response: Argument lists that do not contain enough arguments, argument lists that were generated with the LANGLVL(66) compiler option, and argument lists that were compiled with the LANGLVL(77) compiler option but without character expressions in the argument list must be converted to use the character expressions required and the LANGLVL(77) compiler option.

AFB155I VOPEP : RECL SPECIFIER IS NOT ALLOWED WHEN CONNECTING A SEQUENTIAL ACCESS|KEYED ACCESS FILE, UNIT nn.

AFB155I VOPEP : RECL SPECIFIER IS REQUIRED WHEN CONNECTING A DIRECT ACCESS FILE, UNIT nn.

Explanation: For the first form of the message, the RECL specifier is specified for a sequential file. With the second form of the message, the RECL specifier was not specified for a direct file.

Supplemental Data Provided: nn is the number of the unit specified on the OPEN statement.

Standard Corrective Action: The IOSTAT= variable is set positive and/or the ERR exit is taken. If neither the IOSTAT nor ERR specifier is specified, the program is terminated.

Programmer Response: Correct the program to specify the correct combination of the ACCESS and RECL specifiers.

AFB156I DDCMP : UNABLE TO OBTAIN STORAGE FOR COMMON 'common-name'.

Explanation: There is insufficient storage available to allow allocation for the named common.

Supplemental Data Provided: The name of the common.

Standard Corrective Action: The request is ignored. Processing continues. Any reference to variables in this common will result in termination of this job.

Programmer Response: Rerun the program with larger storage or recompile the program with a smaller common.

AFB157I DDCMP : SHRCOM REQUEST WAS MADE FROM A PARALLEL SUBROUTINE FOR DYNAMIC COMMON aaaaaaaa.

AFB157I DDCMP : AN INVALID VALUE WAS GIVEN FOR THE DYNAMIC COMMON NAME ON THE SHRCOM CALL, aaaaaaaa.

AFB157I DDCMP : SHRCOM ARGUMENT LIST IS IN AN INCORRECT FORMAT.

Explanation:

For format 1 of this message, the SHRCOM service routine was called by a parallel subroutine. A dynamic common block must be made shareable from the main task program.

For format 2 of this message, the dynamic common name was an invalid VS FORTRAN name. The name may not begin with blanks or a digit, and must be 1 to 31 characters long.

For format 3 of this message, the argument list for the SHRCOM service routine specified incorrect data. One of the following conditions was detected:

- Too many or no arguments were specified in the argument list.
- The argument list was not in the format that is generated by the VS FORTRAN compiler when character arguments are provided (the argument list contains other than character expressions).

Supplemental Data Provided:

aaaaaaaa dynamic common name or the invalid value given for the dynamic common name in the SHRCOM request

Standard Corrective Action: The SHRCOM request is ignored and processing continues.

Programmer Response:

For format 1 of this message, remove the SHRCOM request from the parallel subroutine and put it in the main task program.

For format 2 of this message, be sure the character expression for the dynamic common name evaluates to a valid VS FORTRAN name.

For format 3 of this message, ensure that you have only coded one argument and that the argument is in the form of a character expression.

AFB158I DDCMP : LENGTHS OF COMMON 'common-name' ARE NOT CONSISTENT IN ALL MODULES OF THIS PROGRAM.

Explanation: A dynamic common must have the same length in all segments of a program.

Supplemental Data Provided: Name of the common.

Standard Corrective Action: Invocation of a subprogram containing a dynamic common whose length differs from that defined in the calling program will result in termination of this job.

Programmer Response: Specify length of the common to be the same in all segments.

AFB159I BTSHS : SECOND ARGUMENT TO function-name FUNCTION IS INVALID.

Explanation: The second argument is not in the valid range for this bit function.

Supplemental Data Provided: The name of the bit function.

Standard Corrective Action: For ISHFT, the result = 0; for IBSET and IBCLR, the result is the first operand; for BTEST, the result is false.

Programmer Response: Specify the second argument within allowable range.

AFB160I VCOMH : FORMAT NESTED PARENTHESES TABLE OVERFLOW. REDUCE NUMBER OF NESTED PARENTHESES IN PROGRAM AND RECOMPILE.

Explanation: The format contains more nested parentheses than the library table can hold.

Supplemental Data Provided: None.

Standard Corrective Action: Parenthesis group is ignored. Processing continues. Results are unpredictable.

Programmer Response: Reduce the number of parenthesis groups to 50 or less.

AFB161I VASYP : ASYNCHRONOUS I/O IS SUPPORTED ONLY ON THE MVS OPERATING SYSTEM.

Explanation: A program called the asynchronous I/O scheduling routine while running in a CMS environment.

Supplemental Data Provided: None.

Standard Corrective Action: The asynchronous I/O request is ignored and the ARRAY expected to be modified, if a READ (IN#) request, is unchanged. The ARRAY isn't saved or written if it is a WRITE (OUT#) request.

Programmer Response: Run the program on an MVS system, or rewrite the program to use synchronous I/O (unformatted).

AFB162I VVIOS | CVIOS : WRITE STATEMENT CANNOT BE ISSUED TO SEQUENTIALLY ACCESSED VSAM RRDS FILE fffffff.

Explanation: An attempt was made to add a record to a sequentially accessed VSAM relative record file that was not empty when the file was opened.

Supplemental Data Provided: The name of the file (fffffff) upon which the request was made.

Standard Corrective Action: The execution is terminated.

Programmer Response: If a record must be added to a nonempty VSAM relative record file, use the access mode of DIRECT.

AFB163I VVIOS | CVIOS : FILE POSITIONING I/O STATEMENT IS NOT ALLOWED IN THE DIRECT ACCESS MODE.

Explanation: A file positioning input/output statement (REWIND, BACKSPACE, or ENDFILE) was issued to a VSAM direct file.

Supplemental Data Provided: None.

Standard Corrective Action: The execution is terminated.

Programmer Response: Correct the program so that no file positioning input/output statements are issued for VSAM direct files.

AFB164I VVIOS | CVIOS : RECORD LENGTH OF FILE fffffff IS LONGER THAN THE ONE DEFINED IN VSAM CATALOG.

Explanation: The maximum record length for the file found in the VSAM catalog (that is, the value specified in the RECORDSIZE specifier when the VSAM cluster is defined using Access Method Services) is less than the length of the record to be written.

Supplemental Data Provided: The name of the file (fffffff) upon which the request was made.

Standard Corrective Action: The execution is terminated.

Programmer Response: Either correct the program so that the length of the record to be written is not greater than the one in the VSAM catalog, or change the record length in the VSAM catalog by redefining the cluster.

AFB165I VVIOS | CVIOS : FILE fffffff, WHICH IS BEING OPENED FOR SEQUENTIAL ACCESS, MUST BE AN ENTRY SEQUENCED VSAM DATA SET.

AFB165I VVIOS | CVIOS : FILE fffffff, WHICH IS BEING OPENED FOR DIRECT ACCESS, MUST BE A RELATIVE RECORD VSAM DATASET.

Explanation: An attempt was made to open a VSAM file with a file format other than what is required. The correct VSAM file format is listed in the message.

Supplemental Data Provided:

fffffff name of the file for which the OPEN statement was issued.

Standard Corrective Action: The OPEN statement is ignored.

Programmer Response: If you want to use the VSAM KSDS file, you must code ACCESS='KEYED' on the OPEN statement. If you want to use the VSAM linear data set, you must use the data-in-virtual routines. Otherwise, change the data definition statement to refer to a file of the appropriate format (ESDS or RRDS).

AFB166I VVIOS | CVIOS : ENDFILE STATEMENT IS TREATED AS DOCUMENTATION FOR VSAM FILE ffffff.

Explanation: A request was made to write an end-of-file mark on a VSAM or VSAM-managed sequential file.

Supplemental Data Provided: The name of the file (ffffff) for which the request was made.

Standard Corrective Action: The request is ignored.

Programmer Response: Remove the statement after carefully checking the effect of removing the statement.

AFB167I VVIOS | CVIOS : ERROR ON VSAM FILE: ffffff WHEN ATTEMPTING TO PROCESS A(N) xxxxxxxxxx OPERATION RC=yy ERROR CODE=zzz.

Explanation: An error was detected by VSAM while an input or output statement, indicated by xxxxxxxxxx, was being processed. The return code and the error code returned by VSAM were yy and zzz, respectively.

Supplemental Data Provided: The name of the operation that caused the error and the return and error codes from VSAM. ffffff is the name of the file.

Standard Corrective Action: The execution is terminated.

Programmer Response: Determine the cause of the error by examining the VSAM return and error codes.

AFB168I VVIOS | CVIOS : xxxxxxxxxx IS ISSUED TO UNOPENED VSAM FILE ON UNIT nn.

Explanation: An input or output request was made to an unopened VSAM file.

Supplemental Data Provided: The name of the operation (xxxxxxxxxx) issued to an unopened file, and the number of the unit (nn).

Standard Corrective Action: The execution is terminated.

Programmer Response: Make sure that the OPEN statement for the file was successfully executed.

AFB169I CDYNA : FILEDEF FAILED. DISK IS NOT ACCESSED, DISK d. FILE ffffffff.

AFB169I CDYNA : FILEDEF FAILED. ALL DDNAME COMBINATIONS HAVE BEEN EXHAUSTED. FILE ffffffff. FILE MODE fm.

AFB169I CDYNA : FILEDEF FAILED. UNEXPECTED ERROR CODE FROM FSSTATE, ERROR CODE nnn. FILE ffffffff.

AFB169I CDYNA : aaaaaaaa FAILED. UNEXPECTED RETURN CODE FROM FILEDEF, RETURN CODE nnn. FILE ffffffff.

Explanation: An OPEN or INQUIRE statement was being executed for one of the following:

- a CMS file specified by its file identifier (filename, filetype, filemode)
- a scratch file without an explicit file definition

However, an error condition was detected while running under CMS in issuing a file definition or in clearing a file definition.

For format 1 of this message, the disk to which the file identifier refers is not accessed. The disk is the file mode is specified on the FILE specifier in an OPEN statement, or, if the file mode is not specified, is the default disk.

For format 2 of this message, all possible ddname combinations have been issued to the system in trying to generate a ddname for the indicated file.

For format 3 of this message, an error other than file-not-found or disk-not-accessed occurred when an FSSTATE macro instruction was executed.

For format 4 of this message, an unexpected error occurred when a FILEDEF or FILEDEF CLEAR command was issued.

Supplemental Data Provided:

ffffffff	file name or data set name
aaaaaaaa	FILEDEF or FILEDEF CLEAR
d	CMS disk specified in the FILE specifier or the default disk
rrr	record format of CMS file
fm	file mode (CMS)

nnn system return code for FILEDEF or error code for FSSTATE

Standard Corrective Action: The I/O request is ignored and execution continues.

Programmer Response:

For format 1 of this message, access the disk to which the file identifier refers before executing your program.

For format 2 of this message, clear unused or unnecessary file definitions with the formats sDFnnnnn or DFsnnnnn, where s is @, #, or $, and nnnnn is in the range from 00000 to 99999.

For format 3 of this message, a non-zero error code was returned from an FSSTATE macro instruction that was issued internally. The code indicates an error other than file-not-found or disk-not-accessed. For more information on these error codes, refer to *VM/SP Command and Macro Reference*, SC19-6209, or consult with your system programmer.

For format 4 of this message, refer to the information on the error code in *VM/SP Command and Macro Reference*, SC19-6209, or consult with your system programmer.

AFB170I VSIOS : CLOSE STATEMENT NOT ALLOWED FOR ERROR MESSAGE UNIT, UNIT nn.

Explanation: A CLOSE statement was directed to the unit upon which run-time error messages are being directed.

Supplemental Data Provided:

nn unit number of the error message unit

Standard Corrective Action: The CLOSE statement is ignored and execution continues. If the ERR specifier was coded on the CLOSE statement, control is passed to the indicated statement.

Programmer Response: Change the program to request I/O to a unit not being used for error messages.

AFB171I VDIOS | VSIOS : CLOSE WITH STATUS OF KEEP IS NOT ALLOWED FOR A FILE THAT WAS CONNECTED WITH A STATUS OF SCRATCH, FILE fffffff.

Explanation: The file connected to the unit specified in the CLOSE statement was opened as a SCRATCH file and cannot be kept at close time.

Supplemental Data Provided: The name of the file (fffffff) connected to the unit specified in the CLOSE statement.

Standard Corrective Action: The CLOSE status is changed to DELETE and execution proceeds.

Programmer Response: Change either the OPEN or CLOSE STATUS specifier to agree with the file usage.

AFB172I VDIOS | VSIOS : FILE fffffff ALREADY CONNECTED TO A UNIT, OPEN REQUEST IGNORED.

Explanation: A file is already connected to a unit that is different from the unit specified in the OPEN statement.

Supplemental Data Provided: The name of the file (fffffff) specified in the OPEN statement.

Standard Corrective Action: The OPEN request is ignored.

Programmer Response: Change the program to specify a different unit in the OPEN request, or change the logic to use the current unit to which the file is connected.

AFB173I VDIOS | VSIOS : I/O STATEMENT SPECIFYING UNFORMATTED I/O ATTEMPTED ON FORMATTED FILE fffffff.

Explanation: FORMATTED and UNFORMATTED I/O requests are not allowed on the same file.

Supplemental Data Provided: The name of the file (fffffff) for which the request was made.

Standard Corrective Action: The I/O operation is ignored.

Programmer Response: Correct the program to direct FORMATTED and UNFORMATTED I/O to different files.

AFB174I VDIOS | VSIOS : I/O STATEMENT SPECIFYING FORMATTED I/O ATTEMPTED ON UNFORMATTED FILE fffffff.

Explanation: FORMATTED and UNFORMATTED I/O requests are not allowed on the same file.

Supplemental Data Provided: The name of the file (fffffff) for which the request was made.

Standard Corrective Action: The I/O operation is ignored.

Programmer Response: Correct the program to direct FORMATTED and UNFORMATTED I/O to different files.

AFB175I name : I/O OPERATION IGNORED. UNIT NUMBER EXCEEDS THE MAXIMUM ALLOWED FOR UNNAMED FILES, UNIT nnnn.

Explanation: A READ, WRITE or OPEN was attempted for a unit number that is higher than 99 (but not higher than the limit specified for unit numbers when VS FORTRAN was installed at your site).

One of the following caused this message to be issued:

- A READ or WRITE statement referred to a unit number higher than 99 when there was no previous OPEN statement specified for that unit.
- An OPEN statement specified a unit number higher than 99, and there was no FILE specifier.

The limit of the unit number to two digits for unnamed files is due to the default ddname format: FTnnF001 for files being connected for sequential or direct access, and FTnnKkk for files being connected for keyed access, where nn is the unit number, and kk is 01, 02, ..., 99, for each key specified in the KEYS specifier on the OPEN statement.

Supplemental Data Provided:

name	VSIOS, VDIOS, VKIOS, or VOPEP
nnnn	unit identifier

Standard Corrective Action: The statement is ignored, and processing continues.

Programmer Response: Correct the invalid unit number.

AFB180I VINQP | VOPEP : AN INVALID VALUE WAS GIVEN FOR THE FILE SPECIFIER ON THE xx STATEMENT, UNIT nn, FILE fffffff.

Explanation: The FILE specifier on an OPEN or INQUIRE statement specified a name of longer than 8 characters, specified a name that did not start with an alphabetic character, or specified a default file name.

Supplemental Data Provided:

nn	unit number for which the OPEN statement was issued.
xx	OPEN or INQUIRE
ffffffff	file name

Standard Corrective Action: The OPEN statement is ignored.

Programmer Response:· Correct the program to specify a correct file name.

AFB181I VOPEP : AN INVALID VALUE WAS GIVEN FOR THE STATUS SPECIFIER ON THE OPEN STATEMENT, UNIT nn.

Explanation: The STATUS specifier did not specify NEW, OLD, SCRATCH, or UNKNOWN as the status of the file being opened on the unit.

Supplemental Data Provided: The unit number (nn) for which the command was issued.

Standard Corrective Action: STATUS is set to UNKNOWN, and processing continues.

Programmer Response: Correct the program to specify a correct STATUS specifier.

AFB182I VOPEP : AN INVALID VALUE WAS GIVEN FOR THE ACCESS SPECIFIER ON THE OPEN STATEMENT, UNIT nn.

Explanation: The ACCESS specifier did not specify SEQUENTIAL or DIRECT for the type of file access to be employed on the unit.

Supplemental Data Provided: The unit number (nn) for which the OPEN statement was issued.

Standard Corrective Action: The OPEN request is ignored.

Programmer Response: Correct the program to specify a correct ACCESS specifier.

AFB183I VOPEP : AN INVALID VALUE WAS GIVEN FOR THE BLANK SPECIFIER ON THE OPEN STATEMENT, UNIT nn.

Explanation: The BLANK specifier did not specify ZERO or NULL for the treatment of blanks on a FORMATTED I/O request.

Supplemental Data Provided: The unit number (nn) for which the OPEN statement was issued.

Standard Corrective Action: The BLANK specifier is assigned the value NULL.

Programmer Response: Correct the program to specify a correct BLANK specifier.

AFB184I VOPEP : AN INVALID VALUE WAS GIVEN FOR THE FORM SPECIFIER ON THE OPEN STATEMENT, UNIT nn.

Explanation: The FORM specifier did not specify FORMATTED or UNFORMATTED for the file.

Supplemental Data Provided: The unit number (nn) for which the OPEN statement was issued.

Standard Corrective Action: The OPEN request is ignored.

Programmer Response: Correct the program to specify the correct formatting technique.

AFB185I VOPEP : STATUS OF SCRATCH IS NOT ALLOWED WHEN CONNECTING A NAMED FILE, FILE fffffff.

Explanation: An OPEN specified FILE and STATUS='SCRATCH' at the same time. The STATUS value is not allowed.

Supplemental Data Provided: The name of file (fffffff) for which the request was made.

Standard Corrective Action: The STATUS value is set to UNKNOWN and processing continues.

Programmer Response: Correct the program to make the two specifiers consistent with each other.

AFB186I VCLOP : AN INVALID VALUE WAS GIVEN ON THE STATUS SPECIFIER ON THE CLOSE STATEMENT, UNIT nn.

Explanation: The STATUS specifier did not specify KEEP or DELETE, or a STATUS of KEEP was specified on the CLOSE statement for a file that was opened with a STATUS of SCRATCH.

Supplemental Data Provided: The unit number (nn) for which the CLOSE statement was issued.

Standard Corrective Action: The STATUS value is set to DELETE if the file was opened as SCRATCH; otherwise, the status is set to KEEP

Programmer Response: Correct the program to specify the correct status values, or make the status of the OPEN and CLOSE consistent with each other.

AFB187I VSPAP : (program-name) CALLED SUBROUTINE (program-name) WITH AN ARRAY (array-name (l:u,...)) HAVING LOWER BOUND(S) GREATER THAN UPPER BOUND(S).

Explanation: When one program unit called another, the called program unit was found to have an array with at least one dimension with a lower bound greater than the upper bound.

Supplemental Data Provided: The names of the calling and called program units, the name of the array, and the lower (l) and upper (u) bound of each dimension in the array.

Standard Corrective Action: Execution continues, but invalid results are probable if a reference is made to the dimension(s) in error.

Programmer Response: Correct the specification of dimensions whose lower bound is greater than the upper bound.

AFB188I CITFN : ARGUMENT TO CHARACTER FUNCTION GREATER THAN 255 OR LESS THAN 0.

Explanation: A value greater than 255 (highest EBCDIC representation) or a value less than 0 has been specified for the CHAR function.

Supplemental Data Provided: None.

Standard Corrective Action: The function is not evaluated, and execution continues. The value of the character function is unpredictable.

Programmer Response: Specify correct value.

AFB189I INDEX : INVALID LENGTH FOR INDEX OPERAND lll, VALUE = vvv; VALUE SHOULD BE BETWEEN 1 AND 32767.

Explanation: The length specified for the second operand of the index function is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: The length (lll) specified for the operand and its value (vvv).

Standard Corrective Action: The function is not evaluated, and execution continues.

Programmer Response: Specify the correct length.

AFB190I VMOPP : THE ERROR NUMBER nnnn DOES NOT FALL WITHIN THE RANGE OF A KNOWN ERROR OPTION TABLE.

Explanation: An error option table that describes the error number (nnnn) could not be found.

Supplemental Data Provided: The error number (nnnn).

System Action: The request is ignored, and execution continues.

For information on the error handling subroutines, refer to Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.

Programmer Response:

If you incorrectly specified a number that you did not intend, change the error number to fall within the range of entries in an error option table. For VS FORTRAN Version 2, valid | numbers are 96 through 301 (VS FORTRAN Version 2 Library) and 302 thrcugh 899 (user-defined). Refer to the documentation for your auxiliary product for the range of error numbers for any auxiliary product you might have used.

If you specified the correct number and the number falls within the range 302 through 899, the range of standard error option table entries should be extended to include the user error number (nnnn). Refer the problem to the people at your installation who give system support for VS FORTRAN Version 2.

If the number you specified falls within the range of an auxiliary product, make sure your product has been initialized. Refer to the documentation for your auxiliary product for information about initializing it.

AFB191I LXCMP : INVALID LENGTH FOR LEXICAL COMPARE, OPERAND xxx. LENGTH VALUE IS: lll.

Explanation: The length specified for the second operand of the LGE, LGT, LLE, or LLT function is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: The operand (xxx) and its length (lll).

Standard Corrective Action: The function is not performed, and execution continues.

Programmer Response: Specify the correct length.

AFB192I VASYP: ASYNCHRONOUS I/O DDNAME ffffffff IS NOT AVAILABLE FOR USE.

Explanation: The ddname for asynchronous I/O was not associated with a disk or a tape file. Asynchronous I/O will not work correctly on a file of any other type.

Supplemental Data Provided: The ddname (ffffffff) of the incorrect type of file.

Standard Corrective Action: The I/O operation is not performed, and execution terminates with a return code of 20.

Programmer Response: Allocate the ddname to a disk or a tape file.

AFB193I CCMPR : INVALID LENGTH FOR CHARACTER COMPARE, OPERAND xxx. LENGTH VALUE IS: lll.

Explanation: The length of the second operand of a character relational compare (.EQ., .LT., ...) not in the range 1 to 32767, inclusive.

Supplemental Data Provided: The operand (xxx) and its length (lll).

Standard Corrective Action: The function is not performed, and execution continues.

Programmer Response: Specify the correct length.

AFB194I VASYP : ASYNCHRONOUS I/O DDNAME ffffffff, IS LINKED TO xxx. A SEQUENTIAL FILE WITH RECFM=VS IS REQUIRED.

Explanation: The ddname used for asynchronous I/O was determined to be connected to an unusable device type. The only acceptable device types are disk and tape. Terminals, SYSIN, SYSOUT, etc., files are not acceptable.

Supplemental Data Provided: The ddname (ffffffff) of the file on which asynchronous I/O was to be attempted.
xxx may one of the following:

A non-DASD device
A partitioned data set member
A VSAM file

Standard Corrective Action: Execution of the program terminates with a return code of 20.

Programmer Response: Connect the file used for asynchronous I/O to an acceptable device type.

AFB195I CMOVE : CHARACTER MOVE INVALID - TARGET AND SOURCE OVERLAP DESTRUCTIVELY.

Explanation: The storage locations assigned to the target and source are such that source data will be destroyed by the requested assignment.

Supplemental Data Provided: None.

Standard Corrective Action: The assignment is not performed, and execution continues.

Programmer Response: Check storage MAP for storage assignments. Also check EQUIVALENCE statements.

AFB196I CMOVE : TARGET LENGTH FOR CHARACTER MOVE GREATER THAN 32767 OR LESS THAN 1.

Explanation: The length of the target (left of equal variable) is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: None.

Standard Corrective Action: The assignment is not performed, and execution continues.

Programmer Response: Specify the correct length.

AFB197I CMOVE : SOURCE LENGTH FOR CHARACTER MOVE GREATER THAN 32767 OR LESS THAN 1.

Explanation: The length of the source (right of equal expression) is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: None.

Standard Corrective Action: The assignment is not performed, and execution continues.

Programmer Response: Specify the correct length.

AFB198I VMOPP : ATTEMPT TO CHANGE UNMODIFIABLE MESSAGE TABLE ENTRY. MESSAGE NUMBER nnnn.

Explanation: The option table specifies that no changes may be made in this entry, but a change request has been made by use of CALL ERRSET or CALL ERRSTR.

For information on the error-handling subroutines, refer to Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.

Supplemental Data Provided: The message number.

System Action: The request is ignored and execution continues.

Programmer Response: Make sure that no attempt has been made to dynamically alter an unmodifiable entry in the option table.

AFB199I CNCAT : LENGTH FOR CONCATENATION OPERAND GREATER THAN 32767 OR LESS THAN 1.

Explanation: The length of one of the operands of a concatenation operation is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: None.

Standard Corrective Action: The concatenation operation is not performed.

Programmer Response: Specify the correct length.

AFB200I VIIOS : END OF INTERNAL FILE, PROCESSING ENDS.

Explanation: The end of an internal file was reached before the completion of an internal I/O request.

Supplemental Data Provided: None.

Standard Corrective Action: Return to END label if the request is a READ; otherwise, the job is terminated.

Programmer Response: Either keep a counter to avoid exceeding the end of the internal record or file, or insert an END specifier on the READ statement for appropriate transfer of control on end of data.

AFB201I VIOUP : UNFORMATTED VARIABLE SPANNED RECORD IS LONGER THAN THE RECORD LENGTH OF lrecl. THE FILE IS NOT COMPATIBLE WITH NON-FORTRAN ACCESS METHODS. FILE fffffff.

AFB201I VIOUP : UNFORMATTED DIRECT ACCESS DATA IS LONGER THAN THE RECORD LENGTH OF lrecl. THE REMAINING DATA IS TRANSFERRED FROM | TO THE NEXT RECORD. FILE fffffff.

Explanation: Your I/O list items represent a record longer than that defined for the file in your unformatted READ or WRITE statement. For the first format of the message, you are writing a variable spanned record longer that the logical record length (LRECL value). For the second format of the message, you are reading or writing from a direct access file and are specifying more data than can be contained in the fixed-length records in the file.

Supplemental Data Provided: The record length (LRECL) defined for the records in the file and the file name fffffff.

Standard Corrective Procedure: For the first format of the message, a record of the size indicated by your I/O list is written, even though this length exceeds the length defined for the records in the file. If you attempt to read this file using non-FORTRAN access methods, you may encounter unexpected results. For the second format of the message, the next higher numbered record in your direct access file is used to complete the data transfer to or from the items in your I/O list, even though this is in violation of the current FORTRAN standard. For either format of this message, execution then continues with no further indication that an error occurred.

Programmer Response: To prevent this message from being printed, you can do one of the following:

- (On MVS or VM): Omit the LRECL specification
- (On MVS only): Specify LRECL=X
- Increase the record length of your file so it is large enough to hold all the data specified by your I/O list. Note, however, that for the second format of this message, which involves a direct access file, increasing record length means you will be able to write or read from only one direct access record at a time.

AFB202I VCIA4 : PROGRAM CANNOT BE DEBUGGED WITH RELEASE 1 LEVEL OF IAD.

Explanation: You specified DEBUG as an execution-time specifier that causes Interactive Debug to be invoked. However, that program product was found to be at the Interactive Debug Release 1 level, which is not compatible with the current release of the VS FORTRAN Version 2 Library.

Your program was link-edited with VS FORTRAN Version 1 Release 4 for execution in link mode.

Supplemental Data Provided: None.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Remove the DEBUG specifier from your execution-time specifiers so that you will not invoke Interactive Debug. You can then run your program without it. Otherwise, remove references to Interactive Debug Release 1 in your JCL, CLIST, or EXEC so you can use VS FORTRAN Version 2 Interactive Debug with VS FORTRAN Version 2. If this is not successful, refer the problem to the people at your installation who give system support for VS FORTRAN Version 2.

AFB203I IBCOP : INVALID COMBINATION OF INITIAL, TEST, AND INCREMENT VALUE FOR READ/WRITE IMPLIED-DO, FILE fffffff; INIT = xxx, TEST = yyy, INCR = zzz.

Explanation: A READ or WRITE statement with an implied DO had an invalid combination of initial, test, and increment values (I1, I2, and I3, respectively) for one of its levels of nesting:

- I3=0, or
- I2 < I1 and I3 ≤ I2-I1, or
- I1 < I2 and I3 < 0.

Supplemental Data Provided:

fffffff	name of the file used in the READ or WRITE operation.
xxx	initial value
yyy	test value
zzz	increment value

Standard Corrective Action: The implied-DO in the I/O list is ignored, and processing continues.

Programmer Response: Check the statements that set the initial, test, and increment variables.

AFB204I VIOLP : ITEM SIZE EXCEEDS BUFFER LENGTH, FILE fffffff.

Explanation: For a non-complex number, the number is longer than the buffer. For a complex number, half the length of the number plus one (for the comma) is longer than the buffer.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Make sure that the record length specified is large enough to contain the longest item in the I/O list.

AFB205I VASYP : I/O SUBTASK ABENDED.

Explanation: The asynchronous I/O subtask resulted in an abnormal termination.

Supplemental Data Provided: None.

Standard Corrective Action: Processing is terminated.

Programmer Response: Verify that all DD statements are coded correctly and refer to the appropriate data sets. Check all READ and WRITE statements and any END FILE, REWIND, and BACKSPACE statements. Check the system completion code for assistance in determining the type of error that caused abnormal termination. Increase storage size as a possible solution.

AFB206I VCVTH : INTEGER VALUE OUT OF RANGE (nnnnnnnn).

Explanation: An input integer was too large to fit into the integer data item. (The largest integer that can be processed is 2**15-1 for INTEGER*2 and 2**31-1 for INTEGER*4.)

Supplemental Data Provided: The input integer (nnnnnnnn).

Standard Corrective Action: The maximum positive or negative value will be returned for the size (2 or 4 bytes) of the receiving field.

Programmer Response: Make sure that all integer input data used is within the required range for the integer variable size.

AFB207I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 377.

AFB208I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 377.

AFB209I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 377.

AFB210I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 377.

AFB211I VCOMH : ILLEGAL field FORMAT CHARACTER SPECIFIED (character), FILE fffffff.

Explanation: An invalid character has been detected in a FORMAT statement.

Supplemental Data Provided: The field containing the character in error, the character specified, and the file name (fffffff).

Standard Corrective Action: Format field treated as an end of format.

Programmer Response: Make sure that all object-time format specifications are valid.

AFB212I VCOMH : FORMATTED I/O, END OF RECORD, FILE fffffff.

Explanation: An attempt has been made to read or write a record, under FORMAT control, that exceeds the buffer length.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: For a read operation, the remainder of the I/O list is ignored; for a write operation, a new record is started with no control character.

Programmer Response: If the error occurs on input, verify that a FORMAT statement does not define a FORTRAN record longer than the record supplied by the data set. No record to be punched should be specified as longer than 80 characters. For printed output, make sure that no record length is longer than the printer's line length.

AFB213I VCOMH | VIOLP | VIOUP | VASYP | VKIOS : rrrr END OF RECORD, FILE fffffff.

Explanation: For VCOMH: The input list for an I/O statement with a FORMAT specification is larger than the logical record.

Supplemental Data Provided: The name of the file (fffffff) and the operation (rrrr).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Make sure the length of the number of elements in the I/O list matches the length of the number of items in the record.

Explanation: For VIOLP: A list-directed READ statement attempted to read more items from a variable spanned logical record than were present in the record. (This message can be issued only when the record format is variable spanned.)

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Make sure that the number of items in the I/O list and the input data agree. Either delete extra variable names or supply additional logical records.

Explanation: For VIOUP and VASYP: The input list in an I/O statement without a FORMAT specification is larger than the logical record.

Supplemental Data Provided: The name of the file (fffffff) and the operation (rrrr).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Make sure the number of elements in the I/O list matches the number of items in the record.

Explanation: For VKIOS: An attempt was made to read or write more than one record in a keyed file with a single READ, WRITE, or REWRITE statement. For keyed files, only one record may be read or written with a single I/O statement.

Supplemental Data Provided: The name of the file (fffffff) and the operation (rrrr).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Modify your I/O statement and your I/O list so the I/O statement processes only one keyed file record.

AFB214I VSIOS | VASYP : RECORD FORMAT INVALID FOR UNFORMATTED OR ASYNCHRONOUS I/O, FILE fffffff.

Explanation: FOR VSIOS: For unformatted records read or written in EBCDIC sequentially organized data sets, the record format specification must be variable spanned and can be blocked or unblocked. This message appears if the programmer has not specified variable spanned, or if an ASCII tape was specified.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: For non-ASCII output data sets, the record format is changed to variable spanned if variable was not specified, or spanned is added to the record format if either variable or variable blocked was specified.

Programmer Response: Correct the record format to variable spanned.

For VASYP: For unformatted records in an asynchronous I/O operation, the record format specification (RECFM) did not include the characters VS.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: For an input operation, the read request is ignored; for an output operation, VS is assumed.

Programmer Response: Change the record format specification to VS.

AFB215I VCVTH : ILLEGAL DECIMAL CHARACTER (character).

Explanation: An invalid character was found in the decimal input corresponding to an I, E, F, or D format code.

Supplemental Data Provided: The record in which the character appeared.

Standard Corrective Action: 0 replaces the character encountered.

Programmer Response: Make sure that all decimal input is valid. Correct any FORMAT statements specifying decimal input where character input should be indicated. Check FORMAT specifications to ensure that correct field widths are specified.

AFB217I name : END OF DATA SET, FILE fffffff.

Explanation: An end-of-data set was sensed during a READ operation, or during a WRITE operation after an ENDFILE; that is, a program attempted to read or write beyond the end of the data. For a named file, a READ may have been attempted after an end-of-file was encountered and an END path was executed.

Supplemental Data Provided:

name	CVIOS, VSIOS, VASYP, or VVIOS.
fffffff	the file name

Standard Corrective Action: The next file is read, that is, the data set sequence number is increased by 1 in the MVS and VM environments.

Programmer Response: Check all job control statements.

For READ operations, either keep a counter to avoid exceeding the end of record or file or insert an END specifier on the READ statement for appropriate transfer of control on end-of-data-set. On a named file, avoid a READ past an end-of-file after an END path was taken.

For WRITE operations, either remove the ENDFILE or insert a BACKSPACE statement after the ENDFILE to position the file at the beginning of the end-of-file record before extending the file with one or more WRITE operations.

AFB218I name : I/O ERROR, FILE fffffff, cccccccccc ERROR OCCURRED WHILE PROCESSING STATEMENT nnnn.

Explanation: An I/O error occurred, usually for one of the following reasons:

- A permanent I/O error has been encountered.
- For sequential I/O, the length of a physical record is inconsistent with the default block size or the block size specified on the job control statement, or the program has attempted to read an empty file.
- An attempt has been made to read or write a record that is less than 18 bytes long on magnetic tape.
- End-of-tape was encountered while writing a tape file.
- For VM only, the program arrived at the end of the medium.
- For VM only, the user did not use the correct file mode number for a file.

Note: If a permanent I/O error has been detected while writing in the object error unit data set, the error message is written to the programmer either at the terminal or the SYSOUT data set, and job execution is terminated.

If two consecutive I/O operations against the same FORTRAN I/O unit result in error message AFB218I, then upon the third I/O operation, depending on the type of I/O operation performed, error message AFB152I may be issued.

Supplemental Data Provided:

name VSIOS, VASYP, or VDIOS.

fffffff The file name

cccccccccc The type of I/O error

nnnn Statement label number or ISN

The short form gives I/O error, file f, and c. The long form gives I/O error, file f, error occurred, but nnnn is not present.

Standard Corrective Action: The I/O request is ignored. After the traceback is completed, control is returned to the call routine statement designated in the ERR=n specifier of an I/O statement, if that specifier was specified.

Programmer Response: For sequential I/O, make sure that the length of the physical record is consistent with the default or specified block size. Check all job control statements. For VM, make sure the disk is defined with a valid file mode. Make sure that no attempt has been made to read or write a magnetic tape record that is fewer than 18 bytes in length.

AFB219I name : ssssssss FAILED. MISSING OR INVALID CONTROL STATEMENT. SYSTEM COMPLETION CODE ccc-rr. FILE fffffff.

AFB219I CFIST : ssssssss FAILED. DISK NOT ACCESSED. FILE fffffff.

AFB219I CFIST : ssssssss FAILED. TAPE NOT ATTACHED. FILE fffffff.

AFB219I CFIST : ssssssss FAILED. UNEXPECTED FSSTATE ERROR CODE fr. FILE fffffff.

AFB219I CFIST : ssssssss FAILED. EIGHT-CHARACTER FILE IS INVALID WHEN REFERRING TO A VSAM FILE. FILE fffffff.

AFB219I VASYP : IMPLICIT OPEN FAILED. MISSING OR INVALID CONTROL STATEMENT. ASYNCHRONOUS I/O IS NOT AVAILABLE. FILE fffffff.

AFB219I VIADI : IMPLICIT OPEN FAILED. MISSING OR INVALID CONTROL STATEMENT. FILE fffffff.

AFB219I VOPEP | VINQP : ssssssss FAILED. ERROR DETECTED IN PREVIOUS INVOCATION OF FILEINF. FILE fffffff.

Explanation: In all cases except for the last two formats of this message, an internal OPEN macro was either attempted and failed, or was not allowed under the given conditions.

If you receive the last format of this message, then an I/O request other than INQUIRE, OPEN, or CLOSE was issued for a preconnected file.

For format 1 (MVS), the DD statement or ALLOCATE command may have specified an incorrect data set name. When the message indicates a missing or invalid control statement, a file may have been referred to in the program but had no corresponding DD statement or ALLOCATE command.

For format 2 (CMS only), the disk on which the file was to be found was not accessed.

For format 3 (CMS only), the tape drive at the virtual address referred to by the symbolic name (TAPn) for the tape device specified on the FILEDEF command is not attached to your userid.

For format 4 (CMS only), an error (other than disk not accessed) occurred when an FSSTATE macro instruction was executed.

For format 5 (CMS only), the 7-character program ddname specified on the DLBL command is the same as the first 7 characters of the 8-character file name indicated on the FILE specifier of the OPEN statement. If a FILEDEF was issued for the 8-character name, and a DLBL command was issued for the 7-character ddname, the DLBL command takes precedence and it is assumed that the FILEDEF statement refers to a VSAM file.

For format 6, an asynchronous READ or WRITE statement failed due to a missing DD statement. The file indicated cannot be used for asynchronous I/O for the remainder of this program execution.

For format 7, a failure occurred during processing of the VS FORTRAN IAD command, RECONNECT. The failure was probably caused by a missing or invalid operating system file definition statement, or because the file definition statement or DLBL command points to a VSAM file. VS FORTRAN Version 2 does not allow VSAM files to be preconnected.

For format 8, the OPEN or INQUIRE statement was not processed due to errors in the values of the arguments in the FILEINF call.

Supplemental Data Provided:

name	VSIOS, VDIOS, CFIST, or VFIST.
ssssssss	OPEN, CLOSE, INQUIRE, IMPLICIT OPEN (IMPLICIT OPEN indicates that an internally performed OPEN operation failed.)
ccc	System completion code (For an explanation of the system completion code and reason code, refer to *OS/VS Message Library: VS2 System Codes*, GC28-1008, or *MVS/XA Message Library: System Codes*, GC28-1157.)
rr	Reason code
fr	FSSTATE error code
fffffff	File name

Standard Corrective Action: The I/O request is ignored and execution continues.

Programmer Response: Either provide the missing operating system file definition statement or correct any erroneous file definition statement. Refer to *VS FORTRAN Version 2 Programming Guide* for more information.

For ISCII/ASCII data sets on MVS, be sure that the LABEL specifier on the DD statement specifies AL (or NL, provided that the DCB subspecifier OPTCD=Q is also specified). Also, be certain that your operating system permits the use of ASCII data sets.

For format 2 of this message, access the disk which holds or will hold the file indicated in the message text. Be certain the disk is linked in the proper mode.

For format 3 of this message, issue a CP QUERY command on the virtual address for the tape device. Check that the virtual address returned by the query corresponds to the symbolic name used in the FILEDEF command. If necessary, have your VM/SP operator issue the commands to attach the tape drive to your userid at the desired virtual address.

For format 4 of this message, check the FSSTATE error return code. For more information on these error codes, see *VM/SP Command and Macro Reference*, SC19-6209.

For format 5 of this message, be sure that any 8-character names in your program are unique through the first 7 characters. If you intend to refer to a VSAM file, specify a file name of 1 to 7 characters that corresponds to the name on the DLBL command.

For format 6 of this message, refer to the response for missing or invalid control statements. Provide or correct the file definition statements and rerun the job.

For format 7 of this message, use the IAD command SYSCMD to issue the operating system file definition statement to define the missing control statements or to correct the existing control statements. (Refer also to the response for missing or invalid control statements.) If you intend to refer to a VSAM file, the RECONNECT command may not be used. If you did not intend to refer to a VSAM file, and you are on CMS, then use the IAD SYSCMD command to clear the DLBL definition and issue the correct FILEDEF command. If you are on MVS, and the file definition statement points to a VSAM file, use the SYSCMD command to correct the file definition statement. In both environments, if the SYSCMD command is successful, reissue the RECONNECT command.

For format 8 of this message, correct illegal parameter specifiers in the FILEINF call. Refer to message 96 which preceded this message for information on what the errors were.

AFB220I name : UNIT NUMBER OUT OF RANGE, UNIT nn.

Explanation: A unit number exceeds the limit specified for unit numbers when the library was installed.

Supplemental Data Provided:

name	VSIOS, VDIOS, DIOCS, CVIOS, VVIOS, or VASYP
nn	unit identifier

Standard Corrective Action: The statement is ignored, and execution continues.

Programmer Response: Correct the invalid unit number.

AFB221I VIONP : NAME FOUND IN NAMELIST INPUT FILE IS TOO LONG. NAME=name.

Explanation: A NAMELIST dictionary name is longer than 31 characters; a variable name is longer than 31 characters (with VS FORTRAN Version 2 Release 2) or longer than 6 characters (with VS FORTRAN Version 2 Release 1.1 or earlier).

Supplemental Data Provided: The first six or 31 characters of the name specified.

Standard Corrective Action: The remainder of the NAMELIST request is ignored.

Programmer Response: Correct the invalid NAMELIST input variable, or provide any missing delimiters.

AFB222I VIONP : NAME NOT IN NAMELIST DICTIONARY NAME=name.

Explanation: An input variable name is not in the NAMELIST dictionary, or an array is specified with an insufficient amount of data.

Supplemental Data Provided: The name specified.

Standard Corrective Action: The remainder of the NAMELIST request is ignored.

Programmer Response: Make sure that a correct NAMELIST statement is included in the source module for all variable and array names read in using NAMELIST.

AFB223I VIONP : END OF RECORD ENCOUNTERED BEFORE EQUAL SIGN. NAME=name.

Explanation: An input variable name or a subscript has no delimiter.

Supplemental Data Provided: The name of the item.

Standard Corrective Action: The remainder of the NAMELIST request is ignored.

Programmer Response: Make sure that all NAMELIST input data is correctly specified and all delimiters are correctly positioned. Check all delimiters.

AFB224I VIONP : SUBSCRIPT FOR NON-DIMENSIONED VARIABLE OR SUBSCRIPT OUT OF RANGE. NAME=name.

Explanation: A subscript is encountered after an undimensioned input name, or the subscript is too large.

Supplemental Data Provided: The name of the item.

Standard Corrective Action: The remainder of the NAMELIST request is ignored.

Programmer Response: Insert any missing DIMENSION statements, or correct the invalid array reference.

AFB225I VCVTH : ILLEGAL HEXADECIMAL CHARACTER char.

Explanation: An invalid character is encountered on input for the Z format code.

Supplemental Data Provided: A display of the record in which the character appeared.

Standard Corrective Action: 0 replaces the encountered character.

Programmer Response: Either correct the invalid character, or correct or delete the Z format code.

AFB226I VCVTH : REAL VALUE OUT OF RANGE (characters).

Explanation: A real number was too large or too small to be processed by the load module. (The largest number that can be processed is $16^{63} - 1$; the smallest number that can be processed is 16^{-65}.)

Supplemental Data Provided: The field of input characters.

Standard Corrective Action: If the number was too large, the result is set to $16^{63} - 1$. If the number was too small, the result is set to 0.

Programmer Response: Make sure that all real input is within the required range for the number specified. Check the format statement used; trailing blanks may be mistaken for zeros in the exponent.

AFB227I VIOLP : ERROR IN REPEAT COUNT, FILE fffffff.

Explanation: An invalid condition was detected while scanning for a (k*---):

- An invalid character was found at the start of the scan,
- A secondary repeat count was detected while under the control of a primary repeat count, or
- The numeric value of the repeat count was invalid.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Make sure that all repeat counts are correctly specified.

AFB228I VASYP : LAST ITEM IN THE I/O LIST HAS A LOWER ADDRESS THAN THE FIRST ELEMENT, FILE fffffff.

Explanation: An I/O list contained an element having a lower storage address than the first element in the list.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The interrupted instruction is ignored, and execution continues.

Programmer Response: Make sure that all elements in the I/O list are specified in the correct order.

AFB229I VPARM : THE AUTOTASK KEYWORD WAS SPECIFIED BUT THE PROGRAM DOES NOT USE ANY MULTITASKING FACILITY FUNCTIONS. THE AUTOTASK KEYWORD IS IGNORED.

Explanation: The application program does not use any multitasking facility functions, but the AUTOTASK keyword was specified in the PARM parameter. This condition is detected only when the load module for the program was link-edited for execution in link mode.

Supplemental Data Provided: None.

Standard Corrective Action: The AUTOTASK keyword is ignored. The multitasking facility is not initialized and execution of the program continues.

Programmer Response: Remove the AUTOTASK keyword from the PARM parameter to prevent the printing of this message.

AFB230I VSERH : SOURCE ERROR AT ISN nnnn—EXECUTION TERMINATED. THE PROGRAM NAME IS "program-name".

Explanation: An attempt to run a program containing compile errors has been intercepted at the execution of the statement in error.

Supplemental Data Provided: The ISN (nnnn) of the statement in the compiled program that is in error, and the name of the routine or subroutine in which the ISN is located.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Correct the source program statement, and rerun the job.

AFB231I VSIOS : SEQUENTIAL I/O ATTEMPTED ON A aaaaaa FILE. UNIT nn.

AFB231I VDIOS : DIRECT ACCESS I/O ATTEMPTED BEFORE AN OPEN OR A DEFINE FILE.

Explanation: Sequential I/O statements were used for a file that is open for keyed or direct access. A program unit cannot use sequential I/O statements.

Supplemental Data Provided:

nn	unit number
aaaa	direct or keyed

Standard Corrective Action: The I/O request is ignored.

Programmer Response:

- Either include the necessary DEFINE FILE or OPEN statement for direct access or delete the OPEN statement for a sequential file. Make sure that all job control statements are correct.
- Make sure the same file name is not used twice within the same program unit for different types of access.

- If you opened the file for direct access and intend to do direct I/O processing, specify a record number in the READ or WRITE statement.

For a file opened for sequential or keyed access, the READ or WRITE statement must *not* contain a number specification.

AFB232I name : RECORD NUMBER nnnnnn OUT OF RANGE, FILE fffffff.

Explanation: The relative position of a record is not a positive integer, or the relative position exceeds the number of records in the data set.

Supplemental Data Provided: The last 5 characters in the name of the module that issued the message: VDIOS, VVIOS, or CVIOS. The record number (nnnnnn) and the name of the file (fffffff).

Standard Corrective Action: The I/O request is ignored.

Programmer Response: Make sure that the relative position of the record on the data set has been specified correctly. Check all job control statements.

AFB233I VDIOS : RECORD LENGTH GREATER THAN 32760 SPECIFIED, FILE fffffff.

Explanation: The record length specified in the DEFINE FILE or OPEN statement exceeds the capabilities of the system and the physical limitation of the volume assigned to the data set in the job control statement.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The record length is set to 32000.

Programmer Response: Make sure that appropriate parameters of the job control statement conform to specifications in the DEFINE FILE or OPEN statement; the record length in both must be equivalent and within the capabilities of the system and the physical limitations of the assigned volume.

AFB234I DIOCS | VDIOS : ATTEMPT TO USE OBJECT ERROR UNIT AS A DIRECT ACCESS DATA SET, UNIT nn.

Explanation: The data set assigned to print execution error messages cannot be a direct access data set.

Supplemental Data Provided: The unit number (nn).

Standard Corrective Action: The request for direct I/O is ignored.

Programmer Response: Make sure that the object error unit specified is not direct access.

AFB235I VSIOS : DIRECT I/O ATTEMPTED ON A aaaaaaaaaa FILE. UNIT nn.

Explanation: Direct I/O statements were used for a file open for sequential or keyed access. A program unit cannot use direct I/O statements in such a case.

Supplemental Data Provided:

nn is the unit number specified in the I/O statement.

aaaaaaaaaa is either SEQUENTIAL or KEYED.

Standard Corrective Action: The I/O request is ignored.

Programmer Response:

- If you want to do direct I/O processing, statement include the necessary DEFINE FILE or OPEN for direct access.
- Make sure the same file name is not used twice within the same program unit for different types of access.

AFB236I VDIOS : DIRECT ACCESS READ REQUESTED BEFORE FILE WAS CREATED, FILE fffffff.

Explanation: A READ is executed for a direct access file that has not been created.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The I/O request is ignored.

Programmer Response: Make sure that either a file utility program has been used, or appropriate parameters have been specified on the associated job control statement. For further information, see *VS FORTRAN Version 2 Programming Guide.*

AFB237I VDIOS : INCORRECT RECORD LENGTH SPECIFIED, FILE fffffff.

Explanation: The length of the record did not correspond to the length of the record specified in the DEFINE FILE or the OPEN statement.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The I/O request is ignored.

Programmer Response: Make sure that the length of the records supplied matches the length specified in the DEFINE FILE or the OPEN statement. If necessary, change the statement to specify the correct record length.

AFB238I VIOLP : INCORRECT DELIMITER IN COMPLEX OR LITERAL INPUT, FILE fffffff.

Explanation: A literal string in the input record(s) was not closed with an apostrophe (or was longer than 256 characters); alternatively, a complex number in the input record(s) contained embedded blanks, no internal comma, or no closing right parenthesis..

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Supply the missing apostrophe, or amend the literal data to keep within the 256-character limit if the error was in the literal input. Check complex input numbers to see that they contain no embedded blanks, and that they contain an internal comma and a closing right parenthesis.

AFB239I VASYP : BLKSIZE IS NOT SPECIFIED FOR AN INPUT FILE, FILE fffffff.

Explanation: The block size for an input file was not specified in the JCL or was specified as zero.

Supplemental Data Provided: The name of the file (fffffff) for which the error occurred.

Standard Corrective Action: The I/O request is ignored.

Programmer Response: Make sure the block size is specified on the JCL for a new file.

```
AFB240I  VABEX : ABEND CODE IS: SYSTEM
         sss, USER uuu.
         SCB/SDWA=aaaaaaaa
         IO cccccccc.
         PSW=xxxxxxxxxxxxxxxx
         ENTRY POINT=eeeeeeee.
         REGS 0 - 3   nnnnnnnn nnnnnnnn
         nnnnnnnn nnnnnnnn
         REGS 4 - 7   nnnnnnnn nnnnnnnn
         nnnnnnnn nnnnnnnn
         REGS 8 -11  nnnnnnnn nnnnnnnn
         nnnnnnnn nnnnnnnn
         REGS 12-15 nnnnnnnn nnnnnnnn
         nnnnnnnn nnnnnnnn
         FRGS 0 & 2  nnnnnnnn nnnnnnnn
         nnnnnnnn nnnnnnnn
         FRGS 4 & 6  nnnnnnnn nnnnnnnn
         nnnnnnnn nnnnnnnn
         DATA AT ADDRESS (xxxxxxxx)
         yyyyyyyy yyyyyyyy yyyyyyyy.
         DYNAMIC COMMON MAP
         dddddd AT ffffffff (gggggggg)
         ddddddd AT ffffffff(gggggggg)
         MAP FOR SHAREABLE LOAD
         MODULE: bbbbbbbb
         hhhhhhhh AT ffffffff
         hhhhhhhh AT ffffffff
         LOADED LIBRARY MODULES
         jjjjjjjj AT fffffff
         jjjjjjjj AT ffffffff
```

Explanation: An abnormal program termination has occurred. Message AFB240I is printed on the object program error unit and included in the message class for the job. The AFB240I message may be preceded by an AFB210I message that is printed on the object program error unit.

Supplemental Data Provided: sss is the completion code when a system code caused termination; uuu is the completion code when a program code caused termination.

For specific explanations of the completion codes, see the messages and codes manual that applies to your operating system.

The SCB/SDWA field gives the address (aaaaaaaa) of the system diagnostic work area, which contains the old PSW (xxxxxxxxxxxxxxxx) and the contents of the general and floating-point registers at the time of the abend. These fields have been copied from the SDWA into this message.

The status of input/output operations is shown in the field IO cccccccc. The variable part of the field contains the word QUIESCED, HALTED, CONTINUED, or NONE. The meanings of these words are:

> QUIESCED—All I/O operations have been completed; no I/O operation is outstanding.
>
> HALTED—Some I/O operations may not have been completed. If records were being written, you should check that all of them were actually written.
>
> CONTINUED—I/O operations were not completed. The program can continue, but FORTRAN does not allow it.
>
> NONE—No I/O operation was active when the abend occurred.

The ENTRY POINT field gives the entry point address (eeeeeeee) of the module in which the abend occurred.

The following items are controlled by the run-time option ABMODLST.

If dynamic common blocks have been used, a map of obtained COMMON areas is provided where dddddd is the name of the COMMON, ffffffff is the starting address of the COMMON, and gggggggg is the length in hexadecimal. If shareable FORTRAN routines have been loaded, a map of the shareable parts is provided, where bbbbbbbb hhhhhhhh is the shareable part name and ffffffff is the starting address of the executable code. If LOAD MODE has been used, a map of LOADED library modules is provided, where jjjjjjj is the library module name and ffffffff is the address of the module.

The failure point address (xxxxxxxx) and the four bytes of data around the failure point address (yyyyyyyy) are also displayed.

Two more lines can appear at the end of the message. The line **TRACEBACK MAY NOT BEGIN WITH ABENDING ROUTINE** is added if VS FORTRAN Version 2 finds an error in the save-area chain. The line **ABEND OCCURRED IN FORTRAN PROCESSING ORIGINAL ABEND** is added if a second abend occurs during the processing of the original abend. In this case, message AFB240I is issued again, and its contents pertain to the second abend.

If the abending module or any module in the traceback chain was compiled with the SDUMP or TEST options, SDUMP output is produced for the module.

Standard Corrective Action: None.

Programmer Response: Use the abend code, the contents of the SDWA and PSW, and any accompanying system messages, to determine the nature of the error.

AFB241I FIXPI : INTEGER BASE=0, INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation (I**J) in the subprogram AFBFIXPI (FIXPI#), where I and J represent integer variables or integer constants, I is equal to 0 and J is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action: Result = 0.

Programmer Response: Make sure that integer variables and/or integer constants for an exponentiation operation are within the allowable range. If the base and exponent may or will fall outside that range during program execution, then either modify the operands, or insert source code to test for the situation and make the appropriate adjustments. Bypass the exponentiation operation if necessary.

AFB242I FRXPI : REAL*4 BASE=0.0, INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation (R**J) in the subprogram AFBFRXPI (FRXPI#), where R represents a REAL*4 variable or REAL*4 constant and J represents an integer variable or integer constant, R is equal to 0 and J is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=0.0,EXP<0,RESULT=•;
If BASE=0.0,EXP=0,RESULT=1.

Programmer Response: Make sure that both the real variable or constant base and the integer variable or constant exponent for an exponentiation operation are within the allowable range. If the base and exponent may or will fall outside that range during program execution, then either modify the operand(s), or insert source code to test for the situation and make the appropriate adjustments. Bypass the exponentiation operation if necessary.

AFB243I FDXPI : REAL*8 BASE=0.0, INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation (D**J) in the subprogram AFBFDXPI (FDXPI#), where D represents a REAL*8 variable or REAL*8 constant and J represents an Integer variable or Integer constant, D is equal to 0 and J is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action: The• is the correctly signed largest representable floating-point number.

If BASE=0.0,EXP<0,RESULT=•;
If BASE=0.0,EXP=0,RESULT=1.

Programmer Response: Make sure that both the real variable or constant base and the integer variable or constant exponent for an exponentiation operation are within the allowable range. If the base and exponent may or will fall outside that range during execution, then either modify the operand(s), or insert source code to test for the situation and make the appropriate compensation. Bypass the exponentiation operation if necessary.

AFB244I FRXPR : REAL*4 BASE=0.0, REAL*4 EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation (R**S) in the subprogram AFBFRXPR (FRXPR#), where R and S represent REAL*4 variables or REAL*4 constants, R is equal to 0 and S is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=0.0,EXP<0.0,RESULT=•;
If BASE=0.0,EXP=0,RESULT=1.

Programmer Response: Make sure that both the real variable or constant base and exponent for an exponentiation operation are within the allowable range. If the base and exponent may or will fall outside that range during program execution, then either modify the operand(s), or insert source code to test for the situation and make appropriate compensation. Bypass the exponentiation operation if necessary.

AFB245I FDXPD : REAL*8 BASE=0.0, REAL*8 EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation (D**P) in the subprogram AFBFDXPD (FDXPD#), where D and P represent REAL*8 variables or REAL*8 constants, D is equal to 0 and P is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=0.0,EXP<0.0,RESULT=•;
If BASE=0.0,EXP=0,RESULT=1.

Programmer Response: Make sure that both the real variable or constant base and exponent for an exponentiation operation are within the allowable range. If the base and exponent may or will fall outside that range during program execution, then either modify the operand(s), or insert source code to test for the situation and make appropriate compensation. Bypass the exponentiation operation if necessary.

AFB246I FCXPC : COMPLEX*8 BASE = (0.0,0.0), REAL PART OF COMPLEX*8 EXPONENT=exponent, LESS THAN OR EQUAL TO 0.

AFB246I FCXPI : COMPLEX*8 BASE = (0.0,0.0), INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO 0.

Explanation: For an exponentiation operation (Z**P) where the complex*8 base Z equals 0, either in the subprogram AFBFCXPC (FCXPC#), the real part of the COMPLEX*8 exponent P, or in the subprogram AFBFCXPI (FCXPI#), the integer exponent P, is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=0.0,0.0,exponent < 0,RESULT=•+0i;
If BASE=0.0,0.0,exponent=0,RESULT=1+0i

Programmer Response: Make sure that both the base and exponent for an exponentiation operation are within the allowable range during program execution. If the base and exponent may or will fall outside that range during program execution, then either modify the operand(s), or insert source code to test for the situation and make the appropriate compensation. Bypass the exponentiation operation if necessary.

AFB247I FCDCD : COMPLEX*16 BASE = (0.0,0.0), REAL PART OF COMPLEX*16 EXPONENT=exponent, LESS THAN OR EQUAL TO 0.

AFB247I FCDXI : COMPLEX*16 BASE = (0.0,0.0), INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO 0.

Explanation: For an exponentiation operation (Z**P) where the COMPLEX*16 base Z equals 0, either in the subprogram AFBFCDCD (FCDCD#), the real part of the COMPLEX*16 exponent P, or in the subprogram AFBFCDXI (FCDXI#), the integer exponent P, is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=(0.0,0.0)exponent < 0,RESULT=•+0i;
If BASE=(0.0,0.0)exponent=0,RESULT=1+0i.

Programmer Response: Make sure that both the base and exponent for an exponentiation operation are within the allowable range during program execution. If the base and exponent may or will fall outside that range during program execution, then either modify the operand(s), or insert source code to test for the situation and make the appropriate compensation. Bypass the exponentiation operation if necessary.

AFB248I FQXPI : REAL*16 BASE=0.0, INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO 0.

Explanation: For an exponentiation operation (Q**J) in the subprogram AFBFQXPI (FQXPI#), where Q represents a REAL*16 variable or constant and J represents an integer variable or constant, Q is equal to 0 and J is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=0.0,EXP < 0,RESULT=•;
If BASE=0.0,EXP=0,RESULT=1.

Programmer Response: Make sure that both the real variable or constant base and the integer variable or constant exponent for an exponentiation operation are within the allowable range. If the base and exponent may or will fall outside that range during execution, then either modify the operand(s), or insert source code to test for the situation and make the appropriate compensation. Bypass the exponentiation operation if necessary.

AFB249I FQXPQ : REAL*16 BASE=base, REAL*16 EXPONENT=exponent, BASE=0.0 AND EXPONENT LESS THAN OR EQUAL TO 0 OR BASE LESS THAN 0 AND EXPONENT NOT EQUAL TO 0.

Explanation: For an exponentiation operation (X**Y) in the subprogram AFBFQXPQ (FQXPQ#), where X and Y represent REAL*16 variables or constants, if X equals 0, Y must be greater than 0; if X is less than 0, Y must equal 0. One of these conditions has been violated.

Supplemental Data Provided: The base and exponent specified.

Standard Corrective Action:

If BASE=0.0 and EXP<0,RESULT=•;
If BASE=0.0 and EXP=0,RESULT=1;
If BASE= < 0.0 and EXP≠0,RESULT=|X|**Y.

Programmer Response: Make sure that both the real variable or constant base and exponent for an exponentiation operation are within the allowable range. If the base and exponent may or will fall outside that range during program execution, then either modify the operand(s), or insert source code to test for the situation and make appropriate adjustments. Bypass the exponentiation operation if necessary.

AFB250I FQXPQ : REAL*16 BASE=base, REAL*16 EXPONENT=exponent, ARGUMENT COMBINATION EXPONENT*LOG2(BASE) GREATER THAN OR EQUAL TO 252.

Explanation: For an exponentiation operation in the subprogram AFBFQXPQ, (FQXPQ#) the argument combination of $Y*\log_2(X)$ generates a number greater than or equal to 252.

Supplemental Data Provided: The arguments specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the base and exponent are within the allowable range. If necessary, restructure arithmetic operations.

AFB251I SSQRT : ARG=argument, LESS THAN ZERO.

Explanation: In the subprogram AFBSSQRT (SQRT), the argument is less than 0.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=$|X|^{1/2}$.

Programmer Response: Make sure that the argument is within allowable range. Either modify the argument, or insert source code to test for a negative argument and make the necessary adjustments. Bypass the function reference if necessary.

AFB252I SEXP : ARG=argument, GREATER THAN 174.673.

Explanation: In the subprogram AFBSEXP (EXP), the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the exponentiation function is within allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB253I SLOG : ARG=argument, LESS THAN OR EQUAL TO ZERO.

Explanation: In the subprogram AFBSLOG (ALOG and ALOG10), the argument is less than or equal to 0. Because this subprogram is called by an exponential subprogram, if the alternative library is used, this message may also indicate that an attempt has been made to raise a negative base to a real power.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If X=0, RESULT=-•;
If X < 0, RESULT=log|X| or $\log_{10}|X|$.

Programmer Response: Make sure that the argument to the logarithmic function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB254I SSCN : ABS(ARGUMENT)=argument GREATER THAN OR EQUAL TO PI*(218)**

Explanation: In the subprogram AFBSSCN (SIN and COS), the absolute value of an argument is greater than or equal to 2**18 * pi (2**18 * pi=.823 550 E+ 06).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=SQRT(2)/2.

Programmer Response: Make sure that the argument (in radians where 1 radian is equivalent to 57.298°) to the trigonometric sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB255I SATN2 : ARGUMENTS = 0.0.

Explanation: In the subprogram AFBSATN2, when the entry name ATAN2 is used, both arguments are equal to 0.

Supplemental Data Provided: None.

Standard Corrective Action: Result=0.

Programmer Response: Make sure that both arguments do not become 0 during program execution, or are not inadvertently initialized or modified to 0. Provide code to test for the situation and, if necessary, modify the arguments or bypass the source referencing the function subprogram.

AFB256I SSCNH : ARG=argument, GREATER THAN OR EQUAL TO 175.366.

Explanation: In the subprogram AFBSSCNH (SINH or COSH), the argument is greater than or equal to 175.366.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: SINH(X)=±•; COSH(X)=•

Programmer Response: Make sure that the argument to the hyperbolic sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB257I SASCN : ARG=argument, GREATER THAN 1.

Explanation: In the subprogram AFBSASCN (ASIN or ACOS), the absolute value of the argument is greater than 1.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If x > 1.0, ACOS(x) = 0;
If x < -1.0, ACOS(x) = pi;
If x > 1.0, ASIN(x) = pi/2;
If x < -1.0, ASIN(x) = -pi/2.

Programmer Response: Make sure that the argument to the arcsine or arccosine function is between -1 and +1, inclusive. If the argument may or will fall outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB258I STNCT : ARG=argument, (HEX=hexadecimal), GREATER THAN OR EQUAL TO PI*(2**18).

Explanation: In the subprogram AFBSTNCT (TAN or COTAN), the absolute value of the argument is greater than or equal to 2**18*pi (2**18*pi=.823 550E+6).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=1.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.2958°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB259I STNCT : ARG = argument, (HEX = hexadecimal), APPROACHES SINGULARITY.

Explanation: In the subprogram AFBSTNCT (TAN or COTAN), the argument value is too close to one of the singularities (±pi/2, ±3pi/2, ... for the tangent or ±pi, ±2pi, ... for the cotangent).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.2958°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will approach the corresponding singularities for the function during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB260I FQXPR : REAL*16 EXPONENT = exponent, GREATER THAN OR EQUAL TO 252.

Explanation: In the subprogram AFBFQXPR (FQXP2#), the exponent exceeds 2**252.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the exponent is within the allowable range.

AFB261I LSQRT : ARG = argument, LESS THAN ZERO.

Explanation: In the subprogram AFBLSQRT (DSQRT), the argument is less than 0.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=$|X|^{1/2}$.

Programmer Response: Make sure that the argument is within the allowable range. Either modify the argument, or insert source code to test for a negative argument and make the necessary compensation. Bypass the function reference if necessary.

AFB262I LEXP : ARG = argument, GREATER THAN 174.673.

Explanation: In the subprogram AFBLEXP (DEXP), the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the exponential function is within allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB263I LLOG : = argument, LESS THAN OR EQUAL TO ZERO.

Explanation: In the subprogram AFBLLOG (DLOG and DLOG10), the argument is less than or equal to 0. Because the subprogram is called by an exponential subprogram, if the alternative library is used, this message may also indicate that an attempt has been made to raise a negative base to a real power.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If X = 0 ,RESULT=-•;
If X<0,RESULT=log|X| or log$_{10}$ |X|.

Programmer Response: Make sure that the argument to the logarithmic function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB264I LSCN : ABS(ARG) = argument, GREATER THAN OR EQUAL TO PI*(2**50).

Explanation: In the subprogram AFBLSCN (DSIN and DCOS), the absolute value of the argument is greater than or equal to .353 711 870 600 806 396 D+16.

Supplemental Data Provided: None.

Standard Corrective Action: Result=SQRT(2)/2.

Programmer Response: Make sure that the argument (in radians where 1 radian is equivalent to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB265I LATN2 : ARGUMENTS = 0.0.

Explanation: In subprogram AFBLATN2, when entry name DATAN2 is used, both arguments are equal to zero.

Supplemental Data Provided: None.

Standard Corrective Action: Result=0.

Programmer Response: Make sure that both arguments do not become zero during program execution, or are not inadvertently initialized or modified to zero. Provide code to test for the situation and, if necessary, modify the arguments or bypass the source referencing the function subprogram.

AFB266I SCNH : ARG = argument, GREATER THAN OR EQUAL TO 175.366.

Explanation: In the subprogram AFBSCNH (DSINH or DCOSH), the absolute value of the argument is greater than or equal to 175.366.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: DSINH(X)=±•; DCOSH(X)=•

Programmer Response: Make sure that the argument to the hyperbolic sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB267I LASCN : ARG = argument, GREATER THAN 1.

Explanation: In the subprogram AFBLASCN (DASIN or DACOS), the absolute value of the argument is greater than 1.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If x > 1.0 DACOS(x) = 0;
If x < -1.0 DACOS(x) = pi;
If x > 1.0 DASIN(x) = pi/2;
If x < -1.0 DASIN(x) = -pi/2.

Programmer Response: Make sure that the argument to the arcsine or arccosine function is between -1 and +1, inclusive. If the argument may or will fall outside that range during execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB268I LTNCT : ARG = argument, (HEX = hexadecimal), GREATER THAN OR EQUAL TO PI*(2**50).

Explanation: In the subprogram AFBLTNCT (DTAN or DCOTAN), the absolute value of the argument is greater than or equal to 2**50*pi (2**50*pi=.353 711 887 601 422 01D+16).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=1.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB269I LTNCT : ARG = argument, (HEX = hexadecimal), APPROACHES SINGULARITY.

Explanation: In the subprogram AFBLTNCT (DTAN or DCOTAN), the argument value is too close to one of the singularities (±pi/2, ±3pi/2, ... for the tangent; ±pi, ±2pi, ... for the cotangent).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument (in radians where 1 radian is equivalent to 57.295 779 513 1°0) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will approach the corresponding singularities for the function during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB270I FCQCQ : COMPLEX*32 BASE = (0.0,0.0), REAL PART OF COMPLEX*32 EXPONENT = exponent, LESS THAN OR EQUAL TO 0.

AFB270I FCQXI : COMPLEX*16 BASE = (0.0,0.0), INTEGER EXPONENT = exponent, LESS THAN OR EQUAL TO 0.

Explanation: For an exponentiation operation (Z**P) where the COMPLEX*32 base Z equals 0, either in the subprogram AFBFCQCQ (FCQCQ#) the real part of the COMPLEX*32 exponent P, or in the subprogram AFBFCQXI (FCQXI#) the integer exponent P, is less than or equal to zero.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=(0.0,0.0), exponent<0, RESULT=•+0i;
If BASE=(0.0,0.0), exponent=0, RESULT=1+0i.

Programmer Response: Make sure that both the base and exponent for an exponentiation operation are within the allowable range during program execution. If the base and exponent may or will fall outside the range during program execution, then either modify the operand(s), or insert source code to test for the situation and make the appropriate compensation. Bypass the exponentiation operation if necessary.

AFB271I CSEXP : REAL PART OF ARGUMENT = argument GREATER THAN 174.673.

Explanation: In the subprogram AFBCSEXP (CEXP), the value of the real part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•(COS X + iSIN X), where X is the imaginary portion of the argument.

Programmer Response: Make sure that the argument to the exponential function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB272I CSEXP : |IMAGINARY PART OF ARGUMENT| = |argument| GREATER THAN OR EQUAL TO PI*(218).**

Explanation: In the subprogram AFBCSEXP (CEXP), the absolute value of the imaginary part of the argument is greater than or equal to 2**18*pi (2**18*pi=.823 550E+6).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If x is the real part of the argument, then Result=e^{x}+0*i, where e is the base of natural logarithms.

Programmer Response: Make sure that the argument to the exponential function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation, and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB273I CSLOG : ARGUMENT = (0.0,0.0).

Explanation: In the subprogram AFBCSLOG (CLOG), the real and imaginary parts of the argument are equal to zero.

Supplemental Data Provided: None.

Standard Corrective Action: Result=-•+0i.

Programmer Response: Make sure that both the real and imaginary parts of the argument do not become zero during program execution, or are not inadvertently initialized or modified to zero. Provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB274I CSSCN : |REAL PART OF ARGUMENT| = |argument| GREATER THAN OR EQUAL TO PI*(218).**

Explanation: In the subprogram AFBCSSCN (CSIN or CCOS), the absolute value of the real part of the argument is greater than or equal to 2**18*pi (2**18*pi=.823 550E+6).

Supplemental Data Provided: The argument specified. The real part is set to zero and the computations are redone.

Standard Corrective Action: The real part is set to zero and the computations are redone. If argument is x + iy, then

CCOS Result=COSH(y)+0*i;
CSIN Result=0+SINH(y)*i

where y is the imaginary part of the original argument.

Programmer Response: Make sure that the real part of the argument (in radians, where 1 radian is equivalent to 57.2958°) to the trigonometric sine or cosine function is within the allowable range. If the real part of the argument may or will exceed the range during program execution, then provide code to test for the situation and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

AFB275I CSSCN : |IMAGINARY PART OF ARGUMENT| = |argument| GREATER THAN 174.673.

Explanation: In the subprogram AFBCSSCN (CSIN or CCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If imaginary part > 0 (X is real portion of argument):

For sine, result=•/2*(SIN X + iCOS X).
For cosine, result=•/2*(COS X - iSIN X).

If imaginary part < 0 (X is real portion of argument):

For sine, result=•/2*(SIN X - iCOS X).
or cosine, result=•/2*(COS X + iSIN X).

Programmer Response: Make sure that the imaginary part of the argument (in radians, where 1 radian is equivalent to 57.2958°) to the trigonometric sine or cosine function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

AFB276I CQEXP : REAL PART OF ARGUMENT = argument GREATER THAN 174.673.

Explanation: In the subprogram AFBCQEXP (CQEXP), the value of the real part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•(COS X + iSIN X), where X is the imaginary portion of the argument.

Programmer Response: Make sure that the real part of the argument to the exponential function is within the allowable range. If the real part of the argument may or will exceed the range during program execution, then provide code to test for the situation, and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

AFB277I CQEXP : |IMAGINARY PART OF ARGUMENT| = |argument| GREATER THAN PI*(2**100).

Explanation: In the subprogram AFBCQEXP (CQEXP), the absolute value of the imaginary part of the argument is greater than 2**100*pi (2**100*pi=.398 244 181 299 569 74D + 31)

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If x is the real part of the argument, then
Result=e^{x}+0*i, where e is the base of natural logarithms.

Programmer Response: Make sure that the imaginary part of the argument to the exponential function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

AFB278I CQLOG : ARGUMENT = (0.0,0.0).

Explanation: In the subprogram AFBCQLOG (CQLOG), the real and imaginary parts of the argument are equal to 0.

Supplemental Data Provided: None.

Standard Corrective Action: Result=-•+0i.

Programmer Response: Make sure that both the real and imaginary parts of the argument do not become 0 during program execution, or are not inadvertently initialized or modified to 0. Provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB279I CQSCN : |REAL PART OF ARGUMENT| = |argument| GREATER THAN OR EQUAL TO 2**100.

Explanation: In the subprogram AFBCQSCN (CQSIN or CQCOS), the absolute value of the real part of the argument is greater than or equal to 2^{100}

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If the argument is X + iY, for CQSIN, result=0 + DSINH(Y)*i and, for CQCOS, result = DCOSH(Y)+0*i.

Programmer Response: Make sure that the real part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the part of the argument may or will exceed the range during program execution, then provide code to test for the situation and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

AFB280I CQSCN : |IMAGINARY PART OF ARGUMENT| = |argument| GREATER THAN 174.673.

Explanation: In the subprogram AFBCQSCN (CQSIN or CQCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If imaginary part > 0 (X is real portion of argument):

For sine, result=•/2*(SIN X + iCOS X).
For cosine, result=•/2*(COS X - iSIN X).

If imaginary part <0, (X is real portion of argument):

For sine, result=•/2*(SIN X - iCOS X).
For cosine, result=•/2*(COS X + iSIN X).

Programmer Response: Make sure that the imaginary part of the argument (in radians, where 1 radian is equal to 57.2957795131°) to the trigonometric sine or cosine function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

AFB281I CLEXP : REAL PART OF ARGUMENT = argument GREATER THAN 174.673.

Explanation: In the subprogram AFBCLEXP (CDEXP), the value of the real part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•(COS X + iSIN X), where X is the imaginary portion of the argument.

Programmer Response: Make sure that the real part of the argument to the exponential function is within the allowable range. If the real part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

AFB282I CLEXP : |IMAGINARY PART OF ARGUMENT| = |argument| GREATER THAN OR EQUAL TO PI*(2**50).

Explanation: In the subprogram AFBCLEXP (CDEXP), the absolute value of the imaginary part of the argument is greater than or equal to 2**50*pi (2**50*pi=0.353 711 887 601 422 01D+16).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If X is the real part of the x argument, then Result=e+0*i, where e is the base of natural logarithms.

Programmer Response: Make sure that the imaginary part of the argument to the exponential function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation, and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

AFB283I CLLOG : ARGUMENT = (0.0,0.0).

Explanation: In the subprogram AFBCLLOG (CDLOG), the real and imaginary parts of the argument are equal to 0.

Supplemental Data Provided: None.

Standard Corrective Action: Result=-•+0i.

Programmer Response: Make sure that both the real and imaginary parts of the argument do not become 0 during program execution, or are not inadvertently initialized or modified to 0. Provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB284I CLSCN : |REAL PART OF ARGUMENT| = |argument| GREATER THAN OR EQUAL TO PI*(250).**

Explanation: In the subprogram AFBCLSCN (CDSIN or CDCOS), the absolute value of the real part of the argument is greater than or equal to 2**50*pi (2**50*pi = 0.353 711 887 601 422 01D+16).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If the argument is X + iY, for CDSIN, the result=0 + DSINH(Y) +i; for CDCOS, the result=DCOSH(Y)+0*i.

Programmer Response: Make sure that the real part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the part of the argument may or will exceed the range during program execution, then provide code to test for the situation, and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

AFB285I CLSCN : |IMAGINARY PART OF ARGUMENT| = |argument| GREATER THAN 174.673.

Explanation: In the subprogram AFBCLSCN (CDSIN or CDCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If imaginary part >0, (X is real portion of argument):

For sine, result=•/2*(SIN X + iCOS X).
For cosine, result=•/2*(COS X - iSIN X).

If imaginary part <0, (X is real portion of argument):

For sine, result=•/2*(SIN X - iCOS X).
For cosine, result=•/2*(COS X + iSIN X).

Programmer Response: Make sure that the imaginary part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

AFB286I VSIOS | VASYP : ATTEMPT TO ISSUE SYNCHRONOUS AND ASYNCHRONOUS I/O REQUESTS WITHOUT AN INTERVENING REWIND, FILE fffffff.

Explanation: A file that has been using one mode of I/O operations (that is, either synchronous or asynchronous) must be rewound before changing modes. An attempt was made to change the mode without rewinding the file.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The I/O request is ignored, and execution continues.

Programmer Response: Insert a REWIND statement at an appropriate point in the program.

AFB287I VASYP : A WAIT ISSUED WITH NO OUTSTANDING I/O REQUEST, FILE fffffff.

Explanation: A WAIT statement was issued with no corresponding READ or WRITE request.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The WAIT statement is ignored, and execution continues.

Programmer Response: Remove the WAIT statement, or include a corresponding READ or WRITE statement.

AFB288I VASYP : NO WAIT ISSUED FOR AN OUTSTANDING I/O REQUEST FILE fffffff.

Explanation: No WAIT statement was issued for an outstanding READ or WRITE request.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: Execution continues with an implied WAIT.

Programmer Response: Include the WAIT statement, or remove the READ or WRITE statement.

AFB289I QSQRT : NEGATIVE ARGUMENT = argument.

Explanation: In the subprogram AFBQSQRT (QSQRT#), the argument is less than zero.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result = $|x|^{1/2}$

Programmer Response: Make sure that the argument is within the allowable range. Either modify the argument, or insert source code to test for a negative argument and make the necessary compensation. Bypass the function reference if necessary.

AFB290I SGAMA : ARG = argument, (HEX = hexadecimal), LESS THAN OR EQUAL TO 2-252 OR GREATER THAN OR EQUAL TO 57.5744.**

Explanation: In the subprogram AFBSGAMA (GAMMA), the value of the argument is outside the valid range (2**-252 < x < 57.5744).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the gamma function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB291I SGAMA : ARG = argument, (HEX = hexadecimal), LESS THAN OR EQUAL TO ZERO OR GREATER THAN OR EQUAL TO 4.2937*1073.**

Explanation: In the subprogram AFBSGAMA (ALGAMA), the value of the argument is outside the valid range (0 < < 4.2937x10**73).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the ALGAMA function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB292I FQXPR : ARG = argument, GREATER THAN 174.673.

Explanation: In the subprogram AFBFQXPR (QEXP), the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the exponential function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB293I QLOG : ARG = argument, LESS THAN OR EQUAL TO ZERO.

Explanation: In the subprogram AFBQLOG (QLOG and QLOG10), the argument is less than or equal to 0. Because the subprogram is called by an exponential subprogram, this message may also indicate that an attempt has been made to raise a negative base to a real power.

Supplemental Data Provided: The argument specified. **Standard Corrective Action:**

If $X=0$, result=-•; if $X<0$, result=$\log|X|$ or $\log_{10}|X|$.

Programmer Response: Make sure that the argument to the logarithm function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB294I QSCN : ARG = argument, GREATER THAN OR EQUAL TO 2**100.

Explanation: In the subprogram AFBQSCN (QSIN and QCOS), the absolute value of the argument is greater than or equal to 2^{100}.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=SQRT(2)/2.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB295I QATN2 : ARGUMENTS = 0.0.

Explanation: In subprogram AFBQATN2, when entry name QATAN2 is used, both arguments are equal to zero.

Supplemental Data Provided: None.

Standard Corrective Action: Result=0.

Programmer Response: Make sure that both arguments do not become zero during program execution, or are not inadvertently initialized or modified to zero. Provide code to test for the situation and, if necessary, modify the arguments or bypass the source referencing the function subprogram.

AFB296I QSCNH : ARG = argument, GREATER THAN 175.366.

Explanation: In the subprogram AFBQSCNH (QSINH or QCOSH), the absolute value of the argument is greater than (or equal to) 175.366.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: QSINH(X) = ±•; QCOSH(X) = •.

Programmer Response: Make sure that the argument to the hyperbolic sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB297I QASCN : ARG = argument, GREATER THAN 1.

Explanation: In the subprogram AFBQASCN (QARSIN or QARCOS), the absolute value of the argument is greater than 1.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If X < 1.0 QARCOS(X) = 0;
If X < -1.0 QARCOS(X) = pi;
If X > 1.0 QARSIN(X) = pi/2;
If X < -1.0 QARSIN(X) = -pi/2.

Programmer Response: Make sure that the argument to the arcsine or arccosine function is between -1 and +1, inclusive. If the argument may or will fall outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB298I QTNCT : ARG = argument, GREATER THAN OR EQUAL TO 2100.**

Explanation: In the subprogram AFBQTNCT (QTAN or QCOTAN), the absolute value of the argument is greater than or equal to 2**100.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=1.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB299I QTNCT : ARG = argument, APPROACHES SINGULARITY.

Explanation: In the subprogram AFBQTNCT (QTAN or QCOTAN), the argument value is too close to one of the singularities (±pi/2, ±3pi/2, for the tangent; ±pi, ±2pi, for the cotangent).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equivalent to 57.295 779 513 1°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will approach the corresponding singularities for the function during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB300I LGAMA : ARG = argument, (HEX = hexadecimal), LESS THAN OR EQUAL TO 2-252 OR GREATER THAN OR EQUAL TO 57.5744.**

Explanation: In the subprogram AFBLGAMA (DGAMMA), the value of the argument is outside the valid range (2**-252 < x < 57.5744).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the DGAMMA function is within the allowable range. If the argument may or will be outside the range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB301I LGAMA : ARG = argument, (HEX = hexadecimal), LESS THAN OR EQUAL TO 0 OR GREATER THAN OR EQUAL TO 4.2937*1073.**

Explanation: In the subprogram AFBLGAMA (DLGAMA), the value of the argument is outside the valid range (0 < x < 4.2937x10**73).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the DLGAMA function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB900I VEMGN : EXECUTION TERMINATING DUE TO ERROR COUNT FOR ERROR NUMBER nnnn.

Explanation: This error has occurred frequently enough to reach the count specified as the number at which execution should be terminated.

Supplemental Data Provided: The error number.

Standard Corrective Action: No corrective action is implemented.

System Action: The job step is terminated with a completion code of 16.

Programmer Response: Make sure that occurrences of the error number indicated are eliminated.

AFB901I VEMGN : EXECUTION TERMINATING DUE TO SECONDARY ENTRY TO ERROR MONITOR FOR ERROR NUMBER nnnn WHILE PROCESSING ERROR NUMBER nnnn.

Explanation: In a user's corrective action routine, an error has occurred that has called the error monitor before it has returned from processing a previously diagnosed error.

Supplemental Data Provided: The error numbers.

Standard Corrective Action: No corrective action is attempted.

System Action: The job step is terminated with a completion code of 16.

Note: If a traceback follows this message, it may be unreliable.

Programmer Response: Make sure that the error monitor is not called prior to processing the diagnosed error.

Example: A statement such as R=A**B (where A and B are REAL*4) cannot be used in the exit routine for error 252, because FRXPR# uses EXP, which detects error 252.

For information on the error-handling subroutines, refer to Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.

AFB902I VEMGN : ERROR NUMBER nnnn DOES NOT FALL WITHIN THE RANGE OF A KNOWN ERROR OPTION TABLE.

Explanation: A call to an internal VS FORTRAN Version 2 library routine from an auxiliary product resulted in a search that failed to find an error option table containing the error number.

Supplemental Data Provided: The error number (nnnn).

System Action: The request is ignored, and execution continues.

Programmer Response:

If the number falls within the range of valid VS FORTRAN Version 2 error numbers (112 through 999), the internal library call resulted in a search that failed to find an error option table containing the error number. Refer the problem to the people at your installation who give system support for VS FORTRAN Version 2.

If the number falls within the range of an auxiliary product, make sure your product has been initialized. Refer to the documentation for the auxiliary product for information about initializing it.

If the number falls within the range of an auxiliary product and the auxiliary product has been initialized, refer the problem to the people at your installation who give support for the auxiliary product.

For other numbers, refer the problem to the people at your installation who give system support for VS FORTRAN Version 2.

AFB904I name : ATTEMPT TO DO I/O DURING FIXUP ROUTINE FOR AN I/O TYPE ERROR, OR FROM A FUNCTION REFERENCE IN AN I/O STATEMENT, FILE fffffff.

Explanation: The user may not issue another I/O statement or call a routine that issues an I/O statement, when attempting to correct an I/O error or from a function referenced in an I/O statement. A reference to a function is not allowed in the list of an I/O statement.

For information on the error-handling subroutines, refer to Chapter 9, "Extended Error-Handling Subroutines and Error Option Table" on page 315.

Supplemental Data Provided: The last 5 characters in the name of the module that issued the message: VSCOM, IBCOM, VIOFP, VIOCP, VIOUP, or VCOM2. The name of the file (fffffff).

System Action: The job step is terminated with a completion code of 16.

Programmer Response: Make sure that, if an I/O error is detected, the user exit routine does not attempt to execute any FORTRAN I/O statement.

AFB905I VCOM2 : SECONDARY ENTRY INTO MAIN ROUTINE, EXECUTION TERMINATED. "IDENTIFY" GAVE RETURN CODE rr.

| **Explanation:** A user program tried to initialize the VS FORTRAN run-time environment a second time. No program can call a FORTRAN main program or issue an unconditional request to ini-

tialize the VS FORTRAN run-time environment if there is a run-time environment already active.

Supplemental Data Provided: The last line of the message, which contains a return code (rr), is provided only if this condition was detected during execution of the IDENTIFY macro instruction in MVS. The return code is obtained when the MVS IDENTIFY macro instruction is used to identify the entry name #VSFTASK, the task list, to the system.

System Action: The job step is terminated with a completion code of 16.

Programmer Response: Make sure that no routine attempts to reenter the main FORTRAN program.

AFB906I **VEMGN : ERROR NUMBER nnn, LINE NO. ll, REQUESTED BY MODULE mod-name HAS NO MESSAGE SKELETON.**

Explanation: The text for line ll of error message number nnn could not be found in the message skeleton, which is supposed to contain all such text.

Supplemental Data Provided: The error number (nnn), the line number (ll) of the message, and the name *(mod-name)* of the library module that tried to print the message.

Standard Corrective Action: The message is not printed, but execution continues.

Programmer Response: Refer the problem to the people at your installation who give system support for VS FORTRAN Version 2.

AFB915I **CLCIO : MODULE AFBVNREN NOT FOUND. A "GLOBAL LOADLIB" IS NEEDED.**

AFB915I **VLCIO : MODULE AFBVLBCM NOT FOUND. A "STEPLIB" OR "FORTIB" IS NEEDED.**

AFB915I **CLCIO : ABEND nnn LOADING AFBVNREN.**

AFB915I **VLCIO : ABEND nnn-rr LOADING AFBVLBCM.**

Explanation: The load module AFBVLBCM (under MVS) or AFBVNREN (under CMS), which is needed to begin running in load mode, could not be loaded. For the first two forms of the message, either the statement identifying the run-time library in which the module resides was not supplied, or else the module is missing from the library. For the last two forms of the message, the condition indicated by the abend code prevented to module from being loaded.

Supplemental Data Provided: The abend code (nnn) from the VM or MVS system, and for MVS, the reason code (rr) associated with the abend code.

Standard Corrective Action: The program is abnormally terminated.

Programmer Response: For the first two forms of the message, make the run-time library available by supplying one of the following:

Under VM, a CMS GLOBAL LOADLIB command
Under MVS, a DD statement with a ddname of JOBLIB, STEPLIB, or FORTLIB
Under TSO, an ALLOCATE command with a ddname of FORTLIB

For the last tow forms of the message, correct the problem (such as insufficient region size) indicated by the abend. If the problem reoccurs, see the system programmer at your installation who give system support for VS FORTRAN Version 2.

AFB916I **VPARM : THE AUTOTASK KEYWORD IS NOT VALID ON THIS SYSTEM.**

Explanation: The AUTOTASK keyword was specified as an run-time parameter. This keyword requests initialization of the multitasking facility. The multitasking facility is only supported on MVS or MVS/XA systems, but it was requested when running under VM.

Supplemental Data Provided: None.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Remove the AUTOTASK keyword as an run-time parameter or run the application program on an MVS or MVS/XA system.

AFB917I VMPRM : THE AUTOTASK KEYWORD SUBPARAMETER STRING IS MISSING.

AFB917I VMPRM | VPARM : THE AUTOTASK KEYWORD SUBPARAMETER STRING IS INVALID.

Explanation: The subparameters of the AUTOTASK keyword in the PARM parameter of the EXEC statement were either missing or in an incorrect format.

Supplemental Data Provided: None.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Correct the subparameters for the AUTOTASK keyword and rerun the job. The required format for the AUTOTASK keyword is:

```
AUTOTASK(stlmod,nn)
```

where *stlmod* is the name of the MVS load module that contains the application program's parallel subroutines and *nn* is the number of subtasks to be created. *stlmod* must be a valid MVS load module contained in the load library specified by the AUTOTASK DD statement. *nn* must be a decimal number between 1 and 99. No blanks are allowed in the subparameter string.

AFB918I VMMAA : THE AUTOTASK DD STATEMENT IS MISSING OR INVALID.

AFB918I VMMAA : THE AUTOTASK DD STATEMENT DOES NOT SPECIFY A VALID LOAD LIBRARY.

Explanation: The multitasking facility (MTF) detected either a missing AUTOTASK DD statement, or a reference to other than a load library.

Supplemental Data Provided: None.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Provide an AUTOTASK DD statement that refers to a data set containing the parallel subroutines load module. This is the load module whose name is given in the AUTOTASK subparameter of the PARM parameter.

AFB919I VMMAA : THE PARALLEL SUBROUTINE LOAD MODULE 'nnnnnnnn' REQUESTED IN THE AUTOTASK KEYWORD DOES NOT EXIST IN THE LOAD LIBRARY SPECIFIED BY THE AUTOTASK DD STATEMENT.

AFB919I VMMAA : THE PARALLEL SUBROUTINE LOAD MODULE 'nnnnnnnn' REQUESTED IN THE AUTOTASK KEYWORD IS NOT A VALID LOAD MODULE.

AFB919I VMMAA : THE PARALLEL SUBROUTINE LOAD MODULE 'nnnnnnnn' REQUESTED IN THE AUTOTASK KEYWORD IS MARKED NOT-EDITABLE. PARALLEL SUBROUTINES CANNOT BE LOCATED.

AFB919I VMMAA : THE PARALLEL SUBROUTINE LOAD MODULE 'nnnnnnnn' REQUESTED IN THE AUTOTASK KEYWORD DOES NOT CONTAIN LIBRARY MODULE VFEIS#.

AFB919I VMMAA : THE PARALLEL SUBROUTINE LOAD MODULE 'nnnnnnnn' REQUESTED IN THE AUTOTASK KEYWORD DOES NOT HAVE LIBRARY MODULE VFEIS# AS THE ENTRY POINT.

AFB919I VMINI : THE PARALLEL SUBROUTINE LOAD MODULE 'nnnnnnnn' REQUESTED IN THE AUTOTASK KEYWORD DOES NOT HAVE THE SAME LINK/LOAD MODE AS THE MAIN TASK PROGRAM LOAD MODULE.

Explanation: The load module, whose name is given in the AUTOTASK subparameter and which is referred to by the AUTOTASK DD statement, was not available in the proper format for use as a parallel subroutine load module.

Supplemental Data Provided: *nnnnnnnn* is the load module name specified with the AUTOTASK keyword.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Be sure that the AUTOTASK DD statement refers to a load library that contains your parallel subroutine load module. In addition, for the following formats of the message:

Format 1—The load module could not be found in the data set. Provide the correct load module name in your AUTOTASK subparameter.

Format 2—Link-edit your parallel subroutines as a load module in the data set referred to by the AUTOTASK DD statement.

Format 3—Link-edit your parallel subroutines, but do not specify the linkage editor attribute NE.

Format 4—Link-edit your parallel subroutines and include the library routine VFEIS# in the load module.

Format 5—Be sure that you do not have a FORTRAN main program or any other program that overrides the entry point name VFEIS#. Link-edit your parallel subroutines and be sure that you include the library routine VFEIS# in the load module. Also, be sure that VFEIS# is the entry point of the parallel subroutine load module.

Format 6—Link-edit either your main task program load module or your parallel subroutine load module so that both execute in link mode, or both execute in load mode.

AFB920I VMEXP | VMSYP | VMNTP : CALL TO MULTITASKING FACILITY FUNCTION ffffff INVALID. THIS FUNCTION WAS CALLED FROM A PARALLEL SUBROUTINE.

AFB920I VMEXP | VMNTP : CALL TO MULTITASKING FACILITY FUNCTION ffffff INVALID. THIS FUNCTION WAS CALLED WITHOUT ARGUMENTS.

AFB920I VMEXP : CALL TO MULTITASKING FACILITY FUNCTION ffffff INVALID. THIS FUNCTION WAS CALLED WHEN THE MULTITASKING FACILITY WAS NOT ACTIVE.

AFB920I VMEXP : CALL TO MULTITASKING FACILITY FUNCTION ffffff INVALID. THE MULTITASKING FACILITY IS NOT SUPPORTED ON THIS SYSTEM.

Explanation: A call to one of the multitasking facility functions was invalid for the reason given.

Supplemental Data Provided: *ffffff* is DSPTCH or SYNCRO or NTASKS.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Refer to the appropriate format of the message, described below:

Format 1—Change the logic of your parallel subroutine so it does not call DSPTCH, SYNCRO, or NTASKS.

Format 2—Provide the proper arguments for DSPTCH or NTASKS.

Format 3—Specify the AUTOTASK keyword in the PARM parameter of your EXEC statement, which invokes your program.

Format 4—Run your MTF application on an MVS or MVS/XA system.

AFB921I VMEXP : THE PARALLEL SUBROUTINE LOAD MODULE DOES NOT CONTAIN THE PARALLEL SUBROUTINE nnnnnn.

Explanation: The parallel subroutine that you requested in your DSPTCH call was not in your parallel subroutine load module.

Supplemental Data Provided: *nnnnnn* is the name of the parallel subroutine requested.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Verify and correct the following items as necessary:

- That the AUTOTASK DD statement specifies the correct load library.
- That the AUTOTASK keyword in the PARM field specifies the correct load module.
- That the correct parallel subroutine was requested in the call to DSPTCH.
- That the requested parallel subroutine was included when the parallel subroutine load module was link-edited.
- That the subroutine name was more than eight characters long.

AFB922I VMTRM : MULTITASKING FACILITY SUBTASK NO. xx FAILED DURING INITIALIZATION.

AFB922I VMTRM : MULTITASKING FACILITY SUBTASK NO. xx FAILED DURING EXECUTION OF PARALLEL SUBROUTINE nnnnnnnn.

AFB922I VMTRM : MULTITASKING FACILITY

SUBTASK NO. xx ENDED DUE TO STOP STATEMENT OR A CALL TO EXIT DURING EXECUTION OF PARALLEL SUBROUTINE nnnnnn.

Explanation: The multitasking facility subtask number *xx* failed or ended as indicated by the message. Information about the failure can be found in the file specified by the FTERR0*xx* DD statement.

Supplemental Data Provided: *xx* is the subtask number and *nnnnnnnn* is the name of the parallel subroutine.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Look in the file specified by the FTERR0*xx* DD statement to determine the cause of failure. Take corrective action as required.

AFB923I name : sssssss STATEMENT WAS EXECUTED FROM A PARALLEL SUBROUTINE FOR AN UNNAMED FILE ON UNIT nn. I/O IN A PARALLEL SUBROUTINE IS RESTRICTED TO NAMED FILES OR TO SEQUENTIAL OUTPUT ON THE ERROR MESSAGE OR PRINT UNIT.

Explanation: The I/O statement in a parallel subroutine was executed for an unnamed file which was not the error message or print unit.

Supplemental Data Provided:

name	VDIOS, VKIOS, or VSIOS
ssssssss	BACKSPACE, REWIND, ENDFILE, DELETE, READ, OPEN, CLOSE, I/O, DRIECT I/O, KEYED ACCESS I/O, or INQUIRE
nn	device unit number

Standard Corrective Action: Processing terminates with a return code of 16

Programmer Response: Provide the name of the file with a FILE specifier on the OPEN statement.

AFB924I VMMAA : ATTACH OF MULTITASKING FACILITY SUBTASK FAILED. RETURN CODE = dd.

Explanation: The MVS ATTACH macro failed while initializing the multitasking facility.

Supplemental Data Provided: *dd* is the return code from the MVS ATTACH macro.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Report the problem to the people at your installation who give system support for VS FORTRAN Version 2. Supply the return code and a dump of the program if possible.

AFB925I VMMAA : BLDL FOR MULTITASKING FACILITY SUBTASK LOAD MODULE FAILED, RETURN CODE = rtc, REASON CODE = rsc.

Explanation: The MVS BLDL macro used during initialization of the multitasking facility failed. If the return code is 8 and the reason code is 0, a permanent I/O error was detected when the system attempted to search the directory. If the return code is 8 and the reason code is 4, insufficient virtual storage was available.

Supplemental Data Provided: *rtc* and *rsc* are respectively the return code and reason code returned by the MVS BLDL macro.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: If BLDL failed because of to a permanent I/O error, report the problem to the people at your installation who give system support. If BLDL failed because of insufficient virtual storage, specify a larger region for your job.

AFB926I VAMTP : ERROR NUMBER nnnnn IN ERROR OPTION TABLE *XXX*UOPT IS IN THE RANGE OF ERROR OPTION TABLE *YYY*UOPT.

Explanation: An attempt to chain the error option table of an auxiliary product with the component ID "XXX" has failed. The error option table, *XXX*UOPT, defines a range of error numbers for table *YYY*UOPT already in the chain. The conflict begins at error message number nnnnn.

Supplemental Data Provided: The error message number (nnnnn), and the names of the conflicting tables (*XXX*UOPT and *YYY*UOPT).

System Action: The request is ignored, and execution continues.

Programmer Response: Refer the problem to the people at your installation who give system support for VS FORTRAN Version 2 or the auxiliary products used by your program.

AFB927I VMEXP : THE MAIN TASK PROGRAM AND THE PARALLEL SUBROUTINE LOAD MODULE HAVE INCOMPATIBLE ADDRESSING MODES.

Explanation: Your main task program was operating in 31-bit addressing mode when it called DSPTCH to schedule a parallel subroutine, but the parallel subroutine load module indicates 24-bit addressing mode.

Supplemental Data Provided: None.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Correct the addressing mode of either the program unit in the main task program or the parallel subroutine load module so they are compatible.

AFB928I VMINI : THE MAIN TASK PROGRAM LOAD MODULE AND THE PARALLEL SUBROUTINE LOAD MODULE ARE NOT OPERATING WITH THE SAME LEVEL OF THE VS FORTRAN LIBRARY.

Explanation: The load module for the main task and the load module for the parallel subroutines have been link-edited with different release levels of the VS FORTRAN library.

Supplemental Data Provided: None.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Re-link either the main task or the parallel subroutine load modules, or both, so they both use the same release level of the library.

AFB929I VLCIO : INITIALIZATION WAS NOT DONE IN THIS TASK. TCB AT xxxxxxxx.

Explanation: The address of the TCB for the executing task could not be found in the list of TCBs for which initialization has been performed. This initialization occurs in a main task by executing a FORTRAN main program or in a parallel subroutine created by the multitasking facility. You invoke this facility by specifying the run-time parameter AUTOTASK. Unless the multitasking facility is used, VS FORTRAN Version 2 does not support execution of FORTRAN programs in tasks other than the one in which the main program runs.

Supplemental Data Provided: *xxxxxxxx* is the address of the TCB.

Standard Corrective Action: Execution is terminated with an user abend code of U929.

Programmer Response: Change the program to run as a single task, or create parallel subroutines and use the multitasking facility.

AFB930I VMTRM : PROGRAM TERMINATED WHILE MULTITASKING FACILITY SUBTASK NO. dd WAS ACTIVE EXECUTING PARALLEL SUBROUTINE nnnnnn.

Explanation: The main task program terminated while the parallel subroutine *nnnnnn* was executing in the MTF subtask *dd*.

Supplemental Data Provided: *dd* is the subtask number; *nnnnnn* is the name of the parallel subroutine active under subtask *dd*.

Standard Corrective Action: None.

Programmer Response: Determine the reason why the main task program terminated. If this program terminated prior to its normal completion, correct the cause of the termination. If the program terminated normally, add a final call to SYNCRO to ensure that all parallel subroutines have completed execution prior to termination of the program.

AFB931I VMEXP | VMTRM : MULTITASKING FACILITY INTERNAL ERROR CONDITION dd.

Explanation: The multitasking facility has detected an internal error condition.

Supplemental Data Provided: *dd* is the code for the error condition detected.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Report the problem to the people at your installation who give system support for VS FORTRAN Version 2. Supply the error code and a dump of the program if possible.

AFB932I VLCIO : ABEND nnn-rr LOADING #VSFTASK. NO LIBRARY INITIALIZATION WAS DONE.

Explanation: The entry name #VSFTASK, an internally-identified name that is needed to continue execution in load mode, could not be located. This indicates that successful initialization of the run-time environment did not occur before an attempt was made to use some library function.

Supplemental Data Provided: The abend code (*nnn*) and the reason code (*rr*) from the MVS system in response to a LOAD macro instruction that requested the entry name #VSFTASK.

Standard Corrective Action: Execution is terminated with system abend code *nnn*.

Programmer Response: If your main program is not a FORTRAN program, change the logic of your program to call the VS FORTRAN Version 2 Library initialization routine VFEIN# once before you call any FORTRAN subroutines. (A FORTRAN main program, when executed, automatically causes initialization of the run-time environment. No subsequent call to VFEIN# is permitted in this case.)

AFB933I VSPIP : VECTOR COMMON AREA ADDRESS IS ZERO. LINKEDIT HAS FAILED.

Explanation: The supplemental copy of the address of the vector common area is zero. Because the vector common area is needed for proper processing, the program is terminated.

Standard Corrective Action: The user program is terminated.

Programmer Response:

On MVS—Review the link-edit listing and supply access to the necessary library. If you are using the loader, supply access to the necessary library.

On VM—Use the GLOBAL TXTLIB or FILEDEF statement to access the necessary library.

When you are finished rebuilding the load module, rerun the program.

AFB934I VSPIP : SECTION SIZE PASSED DOES NOT MATCH THE SYSTEM VALUE.

Explanation: The compiler-generated section size did not match the section-size value obtained from the operating system. The values must match for proper functioning on the hardware.

Standard Corrective Action: The user program is terminated.

Programmer Response: Determine the correct value of the section size, and pass this value to the compiler, or use the default compilation option for any section size.

AFB935I VSPIP : SPILL AREA CHAIN TERMINATED BEFORE ALL AREAS WERE ASSIGNED. USER PROGRAM HAS DAMAGED THE CHAIN POINTER.

Explanation: The routine to obtain compiler-required spill areas has determined that it has to provide more spill areas than it has in its list. This can only happen when the chain pointer connecting the spill areas has been damaged. This damage only occurs if storage has been inadvertently modified by the user program.

Standard Corrective Action: The user program is terminated.

Programmer Response: The most likely reason for this problem is the generation of an invalid array index. Review the source code for possible bad array indexes. The bad array index will lead to storing of data in the wrong place when the assignment is done.

| **AFB936I CSTIO : FILEDEF FOR FTnnF001 FAILED, EXECUTION TERMINATED.**

| **AFB936I VSTIO : FILE ALLOCATION FOR fffffff FAILED, ERROR CODE xxxx, INFORMATION CODE yyyy. EXECUTION TERMINATED.**

| **AFB936I VSTIO : FILE DEALLOCATION FOR fffffff FAILED, ERROR CODE xxxx,**

INFORMATION CODE yyyy. EXECUTION TERMINATED.

Explanation:

For format 1 of this message, a FILEDEF issued internally by VS FORTRAN to define the indicated standard I/O unit failed.

For format 2 of this message, the allocation for the standard error message and print unit failed. These units are allocated to a terminal if running under TSO, and to SYSOUT=A if running in batch. Refer to *MVS/XA Programming Library: System Macros and Facilities, Volume 1*, GC28-1150, or to *OS/VS2 MVS System Programming Library: Job Management*, GC28-1303, for more information on the error and information codes.

For format 3 of this message, the deallocation for the standard error message and print unit failed. Refer to *MVS/XA Programming Library: System Macros and Facilities, Volume 1*, GC28-1150, for more information on the error and information codes. If the information code yyyy is 0, the text 'INFORMATION CODE yyyy' will not be printed.

Supplemental Data Provided:

nn	external unit identifier
ffffffff	ddname for the standard error message and print units (for example, FT06F001, FTERRnnn, FTPRTnnn)
xxxx	system error code returned by the allocation routine
yyyy	information code returned by the allocation routine

Standard Corrective Action: Execution is terminated.

Programmer Response: Report the problem to the person who provides system support for VS FORTRAN at your site.

AFB937I CLOAD | VLOAD : MODULE xxxxxxxx NOT AVAILABLE IN LINK MODE FOR USE BY PRODUCT MODULE yyyyyyyy.

Explanation: While executing in link mode, the VS FORTRAN Library module yyyyyyyy tried to use the module xxxxxxxx, which was not included as part of the executable program. This problem can occur because library modules from more than one level of the product are included within the executable program or because the modules that are included are a mixture of those needed for link mode or load mode.

Supplemental Data Provided:

xxxxxxxx	the name of the VS FORTRAN Library module that was not available
yyyyyyyy	the name of the VS FORTRAN Library module that tried to load xxxxxxxx because it was not available

Standard Corrective Action: Execution is terminated with a return code of 16

Programmer Response: Be sure that the proper libraries for either link mode or load mode were used when the executable program was created and that all VS FORTRAN Library modules, that is, those whose names begin with AFB, are from the same level of VS FORTRAN.

If the problem recurs, refer the problem to the person who provides system support for VS FORTRAN at your site.

Glossary

This glossary includes definitions developed by the American National Standards Institute (ANSI).

An asterisk (*) to the left of a term indicates that the entire entry is reproduced from the *American National Dictionary for Information Processing*, copyright 1977 by the Computer and Business Equipment Manufacturers Association, copies of which may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

•. • is used in this manual to represent the maximum floating-point value.

A

allocation. See "dynamic allocation."

alphabetic character. A character of the set A, B, C,...,Z. See also "letter."

In VS FORTRAN, the currency symbol ($) is considered an alphabetic character. The lowercase letters a,b,...z are equivalent to their uppercase counterparts.

alphanumeric. A letter or digit.

alphanumeric character set. A character set that contains both letters and digits.

argument. A parameter passed between a calling program and a SUBROUTINE subprogram, a FUNCTION subprogram, or a statement function within the calling program.

arithmetic constant. A constant of type integer, real, double precision, or complex.

arithmetic expression. One or more arithmetic operators and/or arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, array element, or function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator. A symbol that directs VS FORTRAN to perform an arithmetic operation. The arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

array. A nonempty sequence of data.

array declarator. The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension. An array declarator may appear in a DIMENSION, COMMON, or explicit type statement.

array element. A data item in an array, identified by the array name followed by a subscript indicating its position in the array.

array name. The symbolic name of an array.

assignment statement. A statement that assigns a value to a variable or array element. It is made up of a variable or array element, followed by an equal sign (=), followed by an expression. The variable, array element, or expression can be character, logical, or arithmetic. When the assignment statement is executed, the value of the expression to the right of the equal sign replaces the value of the variable or array element to the left.

B

basic real constant. A string of decimal digits containing a decimal point, and expressing a real value.

blank common. An unnamed common block.

C

character. An entity which can be stored in one byte.

character context. Characters within the scope of character constant delimiters (apostrophes) or within a Hollerith constant.

character constant. A string of one or more characters enclosed in apostrophes. The delimiting apostrophes are not part of the constant.

character expression. An expression in the form of a single character constant, variable, array element, substring, function reference, or another expression enclosed in parentheses. A character expression is always of type character.

character operator. A symbol that directs VS FORTRAN to perform a character operation. The only character operator, //, directs VS FORTRAN to perform the concatenation of character strings.

character storage unit. A character datum has one character storage unit in a storage sequence for each character in the datum. In VS FORTRAN, a character storage unit is one byte.

character type. A data type that represents characters. Each character occupies one byte of storage.

common block. A storage area that may be referred to by a calling program and one or more subprograms.

compiler directive. An instruction to the compiler to assist processing of FORTRAN source statements, such as input and display. In VS FORTRAN, INCLUDE and EJECT are the only compiler directives.

complex constant. An ordered pair of signed or unsigned real or integer constants separated by a comma and enclosed in parentheses. The first constant of the pair is the real part of the complex number; the second is the imaginary part.

complex type. An approximation of the value of a complex number, consisting of an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

connected file. A file is connected when it refers to a unit. The terms connected file and connected unit are equivalent.

connected unit. A unit is connected when it refers to a file. The terms connected unit and connected file are equivalent.

constant. An unvarying quantity. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and hexadecimal data.

control statement. Any of the statements used to alter the normal sequential execution of statements, or to terminate the execution of a program. FORTRAN control statements are any of the forms of the GO TO, IF, DO and DO WHILE statements, or the PAUSE, CONTINUE, and STOP statements.

D

data. Constants, variables, and arrays.

data-in-virtual object. An external storage area containing a linear string of permanent data that is created, read or updated by a DIV macro request. The external storage area becomes a data-in-virtual object when it is accessed.

data item. A constant, variable, array element, or character substring.

data set. The major unit of data storage and retrieval consisting of data collected in one of several prescribed arrangements and described by control information to which the system has access. See also "file."

data set reference number. A constant or variable in an input or output statement that identifies a data set to be processed; the unit number.

data type. The properties and internal representation that characterize data and functions. The basic types are integer, real, complex, logical, double precision, and character.

ddname. Data definition name specified on the file definition statement.

deallocation. Disassociation of a data set and the device on which it resides from a program. The data set is no longer available to the FORTRAN program unless another allocation is performed.

default file name. A name of the form **FTnnFmmm**, **FTnnKkk**, **FTERRsss** or **FTPRTsss**, where n, m, k, and s, each represent single digits.

dependence. A relationship among FORTRAN statements, which indicates that the order in which statements are executed is important to the results of a program or subroutine. Dependence occurs when a particular storage location is used more than once, either by successive statements or by a single statement during different iterations of a DO loop.

*** digit.** One of the characters 0 to 9. See "numeric character."

dimension. The attribute of size given to arrays in DIMENSION, COMMON, and specification statements. An array may have 1 to 7 dimensions.

disconnected file. A file that is not connected to an external unit is disconnected. If a file is disconnected, no VS FORTRAN I/O statements except INQUIRE, OPEN, or CLOSE can be successfully executed on that file until an open operation successfully connects the file and a unit.

disconnected unit. A unit that is not connected to a file is disconnected. If a unit is disconnected, no VS FORTRAN I/O statements except INQUIRE, OPEN, or CLOSE can be successfully executed on the file associated with the file until an open operation successfully connects the unit and file.

DO loop. A range of statements executed repetitively by a DO statement. See also "range of a DO."

double-byte character. An entity that requires two character storage units (in a FORTRAN file, a double-byte character occupies two columns).

double-byte character set. Characters represented by a two byte-code, where each byte is in the range X'41' to X'FE', except for the double-byte blank (X'4040') These characters are used for very large character sets such as Chinese, Japanese, and Korean.

DO WHILE loop. A range of statements executed repetitively by a DO WHILE statement. See also "range of a DO WHILE."

DO variable. A variable, specified in a DO statement, that is initialized or increased prior to each execution of the statement or statements within a DO range. It is used to control the number of times the statements within the range are executed. See also "range of a DO."

double precision. The standard name for real data of storage length 8.

dummy argument. A variable within a subprogram or statement function definition with which actual arguments from the calling program or function reference are positionally associated. Dummy arguments are defined in a SUBROUTINE or FUNCTION statement, or in a statement function definition.

dummy procedure. A dummy argument that is identified as a procedure. It may be associated only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure.

dynamic allocation. The process by which data sets or files are allocated during processing of a program, without reference to JCL or other file definition statements. With dynamic allocation, VS FORTRAN (not the user) assigns system resources to a program at the time the program is executed.

E

EBCDIC double-byte character. A double-byte character that has X'42' as the first byte.

empty file. A DASD (non-VSAM) file for which no I/O statement resulting in a data transfer to the file has ever been successfully executed.

executable program. A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures, or both.

executable statement. A statement that moves data, performs an arithmetic, character, logical, or relational operation, or alters the sequential execution of statements.

execution instance. A single execution of a program beginning when the program is dispatched until its execution is terminated.

existing file. A file that has been defined (by a DD statement on MVS systems, an ALLOC command on TSO systems, or a FILEDEF command on CMS systems), and, conceptually, resides on the medium.

existing unit. A valid unit number in FORTRAN's internal unit assignment table, as specified at installation.

explicit file definition. A user-provided statement that describes the characteristics of a file to a user program. See "file definition statement."

expression. A notation that represents a value: a constant or a reference appearing alone, or combinations of constants and/or references with operators. An expression can be arithmetic, character, logical, or relational.

external file. A sequence of records in external storage.

external function. A function defined outside the program unit that refers to it.

external procedure. A SUBROUTINE or FUNCTION subprogram written in FORTRAN or any other language accessible to FORTRAN calls.

external symbol. A control section name, entry point name, or external reference that is defined or referred to in a particular program unit.

F

file. A set of records which can be processed as a single unit after the file is opened. Access to these records is gained through the file definitions (i.e., MVS DD statements or CMS FILEDEFs). If the file is located in internal storage, it is an internal file; if it is on an input/output device, it is an external file.

file connection. A file is connected if it is referred to by a unit.

file definition statement. A statement that describes the characteristics of a file to a user program. For example, the OS/VS DD statement for batch processing, or the FILEDEF command for CMS processing. Note that CMS provides a default file definition statement if none is specified.

file existence. See "existing file."

file reference. A reference within a program to a file. It is specified by a unit identifier.

formatted record. (1) A record, described in a FORMAT statement, that is transmitted, when necessary with data conversion, between internal storage and internal storage or to an external record. (2) A record transmitted with list-directed READ or WRITE statements and an EXTERNAL statement.

FORTRAN-supplied procedure. See "intrinsic function."

function reference. A source program reference to an intrinsic function, to an external function, or to a statement function.

function subprogram. A subprogram invoked through a function reference, and headed by a FUNCTION statement. It returns a value to the calling program unit at the point of reference.

H

hexadecimal constant. A constant that is made up of the character Z, followed by two or more hexadecimal digits.

hierarchy of operations. The relative order used to evaluate expressions containing arithmetic, logical, or character operations.

Hollerith constant. A string of any characters capable of representation in the processor and preceded by *w*H, where *w* is the number of characters in the string.

I

implied DO. An indexing specification (similar to a DO statement, but without specifying the word DO) with a list of data elements, rather than a set of statements, as its range. In a DATA statement, the list can contain integer constants or expressions containing integer constants. In input/output statements, the list can contain integer, real, or double precision arithmetic expressions.

induction variable. A variable in a loop that is altered at only one place within the loop by the addition or subtraction of a constant. The value of an induction variable proceeds through an orderly sequence as the loop is executed. DO-loop indexes are the most common example of induction variables.

integer constant. A string of decimal digits containing no decimal point and expressing a whole number.

integer expression. An arithmetic expression whose values are of integer type.

integer type. An arithmetic data type capable of expressing the value of an integer. It can have a positive, negative, or zero value.

internal file. A sequence of records in internal storage.

intrinsic function. A function, supplied by VS FORTRAN, that performs mathematical, character, logical, or bit-manipulation operations.

*** I/O.** Pertaining to either input or output, or both.

I/O list. A list of variables in an input or output statement specifying which data is to be read or which data is to be written. An output list may also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

L

labeled common. See "named common."

length specification. A source language specification of the number of bytes to be occupied by a variable or an array element.

letter. One of the characters from A, B, ..., Z, or $.

| **linear data set.** A type of VSAM data set in which data is stored such that it can be retrieved or updated in 4096 byte blocks.

list-directed. An input/output specification that uses a data list instead of a FORMAT specification.

logical constant. A constant that can have one of two values: true or false.

logical expression. A combination of logical primaries and logical operators. A logical expression can have one of two values: true or false.

logical operator. Any of the set of operators: .NOT., .AND., .OR., .EQV. or .NEQV.

logical primary. A primary that can have the value true or false. See also "primary."

logical type. A data type that can have the value true or false. See also "data type."

looping. Repetitive execution of the same statement or statements. Usually controlled by a DO or a DO WHILE statement.

M

main program. A program unit, required for execution, that can call other program units but cannot be called by them.

N

name. A string of one to six alphanumeric characters, the first of which must be alphabetic. Also known as a symbolic name. A string of 1 to 31 alphanumeric characters and the underscore (_) character.

named common. A separate common block consisting of variables, arrays, and array declarators, and given a name.

named file. A file whose corresponding file definition name is not a system-assigned default file name.

non-empty file. A file for which at least one output statement, which resulted in data transfer to the file, has been successfully executed on the file.

nonexecutable statement. A statement that describes the characteristics of the program unit, of data, of editing information, or of statement functions, but does not cause an action to be taken by the program.

nonexistent file. A file that has not been defined by a FILEDEF command or by job control statements.

* **numeric character.** One of the digits 0 to 9.

numeric constant. A constant that expresses an integer, real, or complex number.

P

parallel subroutine. A subroutine which is concurrently executed with one or more other subroutines.

preconnected file. A unit or file that was defined at installation time. However, a preconnected file does not exist for a program if the file is not defined by a FILEDEF command or by job control statements.

preconnected unit. A unit is preconnected if it is connected at the instant in which the user program begins to run (before any OPEN statements are given for the file). Preconnected units must refer to non-VSAM, physical sequential files with default file names. A preconnection is broken through the successful execution of an OPEN or CLOSE statement.

predefined specification. The implied type and length specification of a data item, based on the initial character of its name in the absence of any specification to the contrary. The initial characters I through N type data items as integer; the initial characters A through H, O through Z, and $ type data items as real. No other data types are predefined. For VS FORTRAN, the length for both types is 4 bytes.

primary. An irreducible unit of data; a single constant, variable, array element, function reference, or expression enclosed in parentheses.

procedure. A sequenced set of statements that may be used at one or more points in one or more computer programs, and that usually is given one or more input parameters and returns one or more output parameters. A procedure consists of subroutines, function subprograms, and intrinsic functions.

procedure subprogram. A function or subroutine subprogram.

program unit. A sequence of statements constituting a main program or subprogram.

R

range of a DO. Those statements that physically follow a DO statement, up to and including the statement specified by the DO statement as being the last to be executed; also called a "DO loop."

range of a DO WHILE. Those statements that physically follow a DO WHILE statement, up to and including the terminating END DO statement; also called a "DO WHILE loop."

real constant. A string of decimal digits that expresses a real number. A real constant must contain either a decimal point or a decimal exponent and may contain both.

real type. An arithmetic data type, capable of approximating the value of a real number. It can have a positive, negative, or zero value.

record. A collection of related items of data treated as a unit.

recurrence. A group of statements in which every member is dependent on some other statement in the group. The group is said to contain a "cycle" of dependences. The analysis of dependences and recurrences is important in the vectorization of scalar code sequences. See also "dependence."

reduction. An operation that takes one or more vector operands and returns a scalar result. Common examples of reduction operations are vector inner (or "dot") products and vector norm.

relational expression. An expression that consists of an arithmetic expression, followed by a relational operator, followed by another arithmetic expression or a character expression, followed by a relational operator, followed by another character expression. The result is a value that is true or false.

relational operator. Any of the set of operators:

.GT.	greater than
.GE.	greater than or equal to
.LT.	less than

.LE. less than or equal to
.EQ. equal to
.NE. not equal to

S

scalar. (1) A variable that can hold only a single datum. (2) Instructions whose operands are single pieces of data are said to be "scalar."

scale factor. A specification in a FORMAT statement, which changes the location of the decimal point in a real number (and, on input, if there is no exponent, the magnitude of the number).

shift-in character. The character (SI) which indicates the end of double-byte character text. It has an internal representation of X'0F'.

shift-out character. The character (SO) which indicates the beginning of double-byte character text. It has an internal representation of X'0E'.

specification statement. One of the set of statements that provides the compiler with information about the data used in the source program. In addition, the statement supplies the information required to allocate data storage.

specification subprogram. A subprogram headed by a BLOCK DATA statement and used to initialize variables in named common blocks.

statement. The basic unit of a FORTRAN program, that specifies an action to be performed, or the nature and characteristics of the data to be processed, or information about the program itself. Statements fall into two classes: executable and nonexecutable.

statement function. A name, followed by a list of dummy arguments, that is equated to an arithmetic, logical, or character expression. In the remainder of the program the name can be used as a substitute for the expression.

statement function definition. A statement that defines a statement function. Its form is a name, followed by a list of dummy arguments, followed by an equal sign (=), followed by an arithmetic, logical, or character expression.

statement function reference. A reference in an arithmetic, logical, or character expression to the name of a previously defined statement function.

statement label. A number of from one through five decimal digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a DO or a DO WHILE, or to refer to a FORMAT statement.

statement number. See "statement label."

stride. The distance between successive elements of a vector, in units of the array element size. From the definition of vector, it follows that the stride of a vector is a constant.

subprogram. A program unit that is invoked by another program unit in the same program. In VS FORTRAN, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

subroutine subprogram. A subprogram whose first statement is a SUBROUTINE statement. It optionally returns one or more parameters to the calling program unit.

* **subscript.** A subscript quantity or set of subscript quantities, enclosed in parentheses and used with an array name to identify a particular array element.

subscript quantity. A component of a subscript: an integer constant, an integer variable, or an expression evaluated as an integer constant.

In VS FORTRAN, a subscript quantity may also be a real constant, variable, or expression.

T

terminal statement. The statement which indicates the end of a DO loop or a DO WHILE loop. For a DO loop, the terminal statement may be any executable statement except the following: unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO. For a DO WHILE loop, the terminal statement must be an END DO statement.

type declaration. The specification of type for the name of a constant, variable, array, or function by use of an explicit type specification statement.

U

undefined. The condition of a variable or array that is not initialized and not assigned a value.

unformatted record. A record that is transmitted unchanged between internal storage and an external record.

unit. A means of referring to a file in order to use input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

unit connection. A unit is connected if it refers to a file.

unit existence. See "existing unit."

unit identifier. The number that specifies an external unit or an internal file.

unnamed file. A file whose corresponding file definition name is a system-assigned default file name.

V

variable. A data item, identified by a name, that is not a named constant, array, or array element, and that can assume different values at different times during program execution. In FORTRAN, the term *variable* is more restrictive than in other programming languages.

vector. An ordered sequence of elements of an array obtained by subscripting the array in a linear manner. Successive elements of the sequence are spaced equidistant in terms of storage locations within the array.

Index

Special Characters

- (blank) 6
- ' (apostrophe) 6, 20
- . (period) v, 6
- + (plus sign) 6
- ! (exclamation point) 9, 12
- $ (currency symbol) 6
- * (asterisk) 11, 27
- - (minus sign) 6, 9
- / (slash) 6
- , (comma) 6
- ( ) (parentheses)
- _ (underscore) 6
- : (colon) 6
- [ ] (brackets)
- = (equal sign) 6
- " (quotation mark) 6, 9
- • (floating-point value) 439
  - See Glossary for definition

A

- A format code 110
- abend data, post 369
- abnormal termination subroutines
  - SYSABN and SYSABD 287
- ABS/DABS/IABS/QABS 250
  - registers used 349
- ABS/IABS error message 419
- absolute value routines, registers used 349
- absolute value subprograms 250
- ACOS 248
  - error message 420
- ACTION specifier
  - error messages 383
- actual argument
  - array name 27
  - in a function subprogram 122
  - in a subroutine subprogram 209
  - in an ENTRY statement 88
- actual arguments in a function subprogram 122
- AIMAG 251
  - registers used 349
- AINT 251
  - registers used 349
- ALGAMA
  - See GAMMA/ALGAMA
- allocation of a file, dynamic 137, 152
- ALOG/ALOG10 246
  - error message 419
  - registers used 349
- alphabetic character
  - See letter
- alphabetic primary
  - See primary
- alternate return parameter 88
- alternative mathematical library subroutines
  - documentation 263
  - register usage 348
- AMOD/DMOD 251
  - registers used 349
- ANINT 252
  - registers used 349
- ANSI definitions 439
- apostrophe 6
  - with character constant 20
- arccosine routines, registers used 348
- arcsine and arccosine subprograms 248
- arcsine routines, registers used 348
- arctangent routines, registers used 348
- arctangent subprograms 248
- argument
  - actual
    - in ENTRY statement 88
    - in subroutine subprogram 209
  - dummy
    - in ENTRY statement 88
    - in subroutine subprogram 209
  - number 244
  - range 244
  - type 244
- argument vector registers 352
- arguments
  - assembler language 346
- arithmetic assignment statement 52
  - conversion rules
    - complex 53
    - integer 53
    - real 53
- arithmetic constant
  - See also digit
  - complex 18
  - integer 15
  - primary 32
  - real 16
- arithmetic expression 32, 340
  - examples 36
  - rules for constructing 32
  - type and length of
    - complex 36
    - integer 35
    - real 35
  - use of parentheses in 34
- arithmetic IF statement 127
- arithmetic operation
  - addition 32
  - division 32
  - evaluation of functions 33

- arithmetic operation *(continued)*
  - exponentiation 32
  - first operand
    - complex 34
    - integer 34
    - real 34
  - multiplication 32
  - order of computation 33
  - subtraction 32
  - unary 32
- arithmetic operator
  - in logical expression 42
  - kinds of 32
- arithmetic routines (modular), registers used 348
- array 24
  - actual argument 26, 27
  - assumed-size 27
  - character 28
  - declarator 26
  - definition 24
  - DIMENSION statement 76
  - dimensions 26, 76
  - dummy argument 27
  - elements 24
  - examples 25
  - output format 358
  - size and type declaration 26
  - subscripts 25
  - upper dimension bound as asterisk 27
- array dimension error message 403
- array items, sample program and output 364
- ASIN 248
  - error message 420
  - registers used 349
- assembler language
  - calling sequence 344
- assign a name to a main program 162
- ASSIGN statement 51
- assigned GO TO statement 125
- ASSIGNM subroutine
  - examples 278
  - rules 277
- assignment statement
  - arithmetic 52
  - character 54
  - logical 54
- associating in Data-in-Virtual 298, 302
- assumed-size array 27
- asterisk 6
- asynchronous
  - READ statement 163
  - WRITE statement 216
- asynchronous I/O error message 399, 404
- asynchronous input/output 339
- AT statement
  - debugging 56
  - in debug packet 72
- ATAN 248
  - registers used 349
- ATAN/ATAN2 error message 420
- ATAN2 248
- automatic function selection 341

B

- BACKSPACE statement 57
- basic real constant 16
- bit function error message 398
- bit functions, explicitly called 345
- bit manipulation functions 257
- bit manipulation routines
  - assembler information 350
  - manipulation routines 269
- blank 6
  - common blocks 67
  - format code 114
  - named common blocks 67
- blank FORMAT statement 114
- blanks, leading 99
- BLOCK DATA statement 59
- block IF statement
  - ELSE 129
  - ELSE IF 130
  - END IF 129
- BN format code 114
- BTEST 257
- BTSHS error message 398
- BZ format code 114

C

- CABS 250
  - registers used 349
- CALL macro instruction 344
  - for vector intrinsic functions 351
- CALL statement 60
  - ASSIGNM 277
  - CDUMP/CPDUMP 273
  - CLOCK 278
  - CLOCKX 278
  - CPUTIME 280
  - DATIM 282
  - DATIMX 282
  - DIVCML 303
  - DIVINF 296
  - DIVINV 300
  - DIVRES 305
  - DIVSAV 305
  - DIVTRF 299
  - DIVTRV 304
  - DIVVWF 298
  - DIVVWV 302
  - DUMP/PDUMP 271
  - DVCHK 270
  - ERRMON 316
  - ERRSAV 317
  - ERRSET 318
  - ERRSTR 321

CALL statement *(continued)*
  ERRTRA 321
  EXIT 283
  FILEINF 283
  OVERFL 270
  SDUMP 274
  SYSABD 287
  SYSABN 287
  SYSRCS 276
  SYSRCT 276
  SYSRCX 276
  UNTANY 288
  UNTNOFD 288
  XUFLOW 291
calling routines in assembler language 345
CCMPR# 266
  error message 404
CDABS 250
CDDVD#/CDMPY# 265
  registers used 349
CDMPY#
  See CDDVD#/CDMPY#
CDSIN/CDCOS 248
  error message 426
  registers used 349
CDVD#/CMPY# 265
  registers used 349
CEXP/CDEXP 246
  error messages 423, 425
  registers used 349
change options 318
CHAR function 256
CHAR specifier
  on INQUIRE statement 143
  on OPEN statement 154
character array element
  READ statement
    direct access, formatted 167
character array name
  READ statement
    direct access, formatted 167
character assignment statement 54
character constant
  definition 19
  length of 20
  mixed 19, 30
  READ statement
    direct access, formatted 166
  transmission 111
  uses of 19
character data dump 273
character data transmission 110
character data type 339
character expression
  operators 37
  READ statement
    direct access, formatted 167
  use of parentheses in 38
character manipulation functions 256
character manipulation routines
  assembler information 350
character operators in logical expressions 42
character set, collating sequence 5
character skipping 111
character substring 28
CHARACTER type statement 91
character variable
  READ statement
    direct access, formatted 167
  storage length 22
  substring 28
characters
  double-byte 6
  source statement 5
  special 5
CHARLEN compiler option 94
CI compiler option 134
CITFN error message 403
CLCI0 error message 431
CLEXP error message 425
CLLOG error message 426
CLOCK subroutine 278
CLOCKX subroutine 278
CLOG/CDLOG 246
  error messages 423, 426
  registers used 349
CLOSE statement 64
  error messages 401, 403
CLSCN error messages 426
CMOVE error messages
  source length for character move invalid 405
  target length for character move invalid 405
CMOVE# 266
  error message 405
CMPLX 253
CMPY#
  See CDVD#/CMPY#
CNCAT# 266
  error message 405
coding the user exit routine 319
colon 6
colon format code 116
comma 6
comma in formatted I/O 99
comments
  definition 8
  fixed-form 11, 13
  free-form 9, 12
  in-line 12, 13
  valid placement 13
common block 67
  named 67
common block to initialize variables 59
common logarithmic routines, registers used 348
COMMON statement 66, 68
compare complex numbers 265

compiler directives
  EJECT statement 48
  INCLUDE statement 48
compiler-directed statement
  EJECT 83
  INCLUDE 134
complex constant 18
  examples 18
complex data type 340
COMPLEX format 271, 357
complex multiply and divide subprograms
  CDDVD#/CDMPY# 265
  CDVD#/CMPY# 265
  CQMPY#/CQDVD# 265
COMPLEX type statement 91
complex variable, storage length 22
complex-to-integer 265
compop 266
computed GO TO statement 126
concatenation
  operation 38
  with double-byte characters 38
concatenation operand error message 405
CONJG 251
  registers used 349
conjugate of a complex number routines, registers used 348
connected file
  BACKSPACE statement 58
  definition 440, 441
  determining properties of a 136
  ENDFILE statement 85
  general description 46
  INQUIRE statement 136
  OPEN statement 151
  preconnection 47
  READ statement
    formatted with direct access 168
    formatted with keyed access 171
    list-directed with external devices 180
    NAMELIST with external devices 184
    unformatted with keyed access 191
    unformatted with sequential access 194
  REWIND statement 198
  REWRITE statement 199
  WRITE statement
    formatted with direct access 219
    formatted with keyed access 221
    formatted with sequential access 224
    list-directed with external devices 229
    NAMELIST with external devices 234
    unformatted with direct access 237
    unformatted with keyed access 239
    unformatted with sequential access 241
connected unit
  See also connected file
  definition 440, 444
  general description 46
connection
  file
    definition 441
    determining 136
  file/unit
    general description 46
  unit
    definition 444
    determining 136
considerations when using debug 72
constant
  arithmetic 15
  assign a name to 158
  character 19
  classes 15
  complex 18
  definition 15
  hexadecimal 21
  Hollerith 20
  integer 15
  logical 19
  mixed character 19
  real 16
constant name, assigning 158
continuation line
  fixed-form 11
  free-form 10
CONTINUE statement 68
continued line, free-form 9
control flow information 369
  output format 359
control statement 45
  assigned GO TO 125
  CALL 60
  computed GO TO 126
  CONTINUE 68
  DO 77
  DO WHILE 82
  ELSE 129
  ELSE IF 130
  END 83
  END DO 84
  ENDIF 129
  execution 45
  GO TO 125
  IF 127
  PAUSE 159
  RETURN 196
  STOP 207
  unconditional GO TO 126
conversion functions 253
conversion rules 53
  data 97
corrective action
  after error 325
  after mathematical subroutine error 331
  after program interrupt 330
COS 248
  error message 419

COTAN 249
CPUTIME subroutine 280
CQABS 250
  registers used 349
CQDVD#
  See CQMPY#/CQDVD#
CQEXP 246
  error messages 424
  registers used 349
CQLOG 246
  error message 425
  registers used 349
CQMPY#/CQDVD# 265
  registers used 349
CQSCN error messages 425
CQSIN/CQCOS 248
  error message 425
  registers used 349
CQSQRT 251
  registers used 349
create a file/preconnected file 151
CSEXP error messages 423
CSIN/CCOS 248
  error message 423
  registers used 349
CSLOG error message 423
CSQRT/CDSQRT 251
  registers used 349
CSSCN error messages 423
CSTIO error messages 437
currency symbol 6
CVIOS error messages
  end of data set 409
  ENDFILE statement 400
  file positioning I/O statement in direct access mode 399
  I/O request made to unopened VSAM file 400
  I/O statement 400
  key sequenced data set 399
  OPEN statement ACTION specifier conflict 387
  record addition conflict 399
  record length too long 399
  record number 414
  unit number out of range 411
CXMPR# 265
  registers used 349

D

D format code 102
DABS 250
  registers used 349
DACOS 248
  error message 422
DASIN 248
  error message 422
  registers used 349
data
  conversion rules 97
data *(continued)*
  definition 15
  kinds of 15
DATA statement 46, 69
  character data in 69
  implied DO in 80
data transfer 112
data type
  character 339
  complex 340
  double precision 342
  extended precision 342
  integer 341
  logical 341
  real 342
data types
  field widths needed 230
data types and lengths 22
data types and valid lengths 22
Data-in-Virtual subroutines
  error messages 394, 397
  fixed-view 296
  Multitasking Facility, using with 314
  page-alignment requirements 313
  varying-view 300
data, double-byte
  moving 277
DATAN 248
  error message 421
  registers used 349
DATAN2 248
  error message 421
date subroutines
  DATIM 282
  DATIMX 282
DATIM subroutine 282
DATIMX subroutine 282
DBCS
  with INQUIRE 143
  with OPEN 154
DBCS compiler option 10, 12
DBCS names 8, 24
DBCS support 10, 154
DBLE 253
DCONJG 251
  registers used 349
DCOS 248
  error message 421
DCOTAN 249
  error message 422
DDCBP error messages 398
DDCMP error message 398
DDIM 252
  registers used 349
ddnames
  default naming conventions 139
  for named/unnamed files 46
DEBUG statement 46, 71
  AT statement 56, 72

- DEBUG statement *(continued)*
  - DISPLAY statement 72, 77
  - END DEBUG statement 72, 84
  - examples 73
  - SUBCHK function 72
  - TRACE OFF statement 72, 213
  - TRACE ON statement 72, 213
- debugging
  - starting 56
- decimal point in format codes 99
- default ddname conventions 139
- default options 325
- define values of
  - array elements
    - DATA statement 69
    - EXPLICIT type statement 91
  - arrays
    - DATA statement 69
    - EXPLICIT type statement 91
  - substrings 69
  - variables
    - DATA statement 69
    - EXPLICIT type statement 91
- DELETE specified on CLOSE statement 64
- DELETE statement 75
- deleting records 75
- DERF/DERFC 251
  - registers used 349
- DEXP 246
  - error message 421
  - registers used 349
- DGAMMA/DLGAMA 251
  - error message 429
  - registers used 349
- digit 6
- DIM 252
  - registers used 349
- DIMAG 251
  - registers used 349
- dimension boundary
  - specified as asterisk 27
- DIMENSION statement 76
  - array declaration 26
  - examples 26
- dimensions, object-time 28
- DINT 251
  - registers used 349
- DIOCS error messages
  - unit number out of range 411
- DIOCS/VDIOS error message 414
- direct access
  - error messages 379, 413
  - files 155
  - READ statement
    - formatted 166
    - unformatted 187
  - WRITE statement
    - formatted 218
    - unformatted 236
- direct access input/output 340
- directives, compiler 48
- disconnect an external file 64
- DISPLAY statement 77
  - in debug packet 72
- DIV facility
  - See Data-in-Virtual subroutines
- DIVCML subroutine 303
- divide complex numbers 265
- divide-check exception test 270
- divide-check subroutine
  - See also DVCHK
  - error message 378
- DIVINF subroutine 296
- DIVINV subroutine 300
- DIVRES subroutine 305
- DIVSAV subroutine 305
- DIVTRF subroutine 299
- DIVTRV subroutine 304
- DIVWWF subroutine 298
- DIVWWV subroutine 302
- DLOG/DLOG10 246
  - error message 421
  - registers used 349
- DMOD
  - See AMOD/DMOD
- DNINT 252
  - registers used 349
- DO loop
  - See also range of a DO
  - nested 79
- DO statement
  - labeled 77
  - unlabeled 77
- DO WHILE loop 82
- DO WHILE statement
  - labeled 82
  - unlabeled 82
- DO-loop 77
- documentation of IBM extensions vi
- double precision
  - data editing 102
  - storage length 22
  - type 91, 133
- double precision data type 342
- DOUBLE PRECISION type statement 91
- double-byte character
  - moving strings 277
- double-byte character data
  - documentation v
- double-byte character set 11
  - character constant 19
  - in continued source lines 10, 12
  - names 8
  - representation 6
  - with INQUIRE 143
  - with OPEN 154
- DP
  - assign 53

DP *(continued)*
extend 54
float 54
DPROD 252
registers used 349
DSIGN 252
registers used 349
DSIN 248
error message 421
registers used 349
DSINH/DCOSH 250
error message 422
registers used 349
DSPAN#/DSPN2# 267
assembler language requirements 350
DSPTCH subroutine 336
DSQRT 251
error message 421
registers used 349
DTAN 249
error message 422
registers used 349
DTANH 250
registers used 349
dummy argument
array name 27
in a function subprogram 123
in a subroutine subprogram 209
in an ENTRY statement 88
dummy arguments in a function subprogram 123
dump an area of storage 271, 273
DUMP/PDUMP subroutine 271
assembler language requirements 350
format specifications 271
output 271
programming considerations 271
dump, symbolic
See symbolic dump 274
dumping storage 271
DVCHK subroutine 270
DYCMN# 267
error message 397
dynamic file allocation 47
error messages 437
with FILEINF subroutine 283
with INQUIRE statement 137
with OPEN statement 152

E

E format code 102
EBCDIC double-byte character 6, 8
EBCDIC names 8
EJECT statement 48, 83
ELSE IF block 130
ELSE IF statement 130
ELSE statement 129
END DEBUG statement 84
in debug packet 72
END DO Statement 84
in a DO loop 84
in a DO WHILE loop 84
end execution error message 429
END IF statement 129
end of data set error message 409
end of line comments 13
end page 98
END statement 83
in a function subprogram 84
in a subroutine subprogram 84
end-of-line comment 11
ENDFILE statement 85
error message 400
ENTRY statement
equal sign 6
EQUIVALENCE statement 89
ERF/ERFC 251
registers used 349
ERR parameters honored for I/O errors 325
ERRMON subroutine 316
error function routines, registers used 348
error function subprograms 251
error handling subroutines 315
error message unit 401
error messages
execution 379
library 375
operator 376
program interrupt 375
error monitor routine 316
error option table 315
error-handling subroutines 270
error, corrective action after 325
ERRSAV subroutine 317
ERRSET subroutine 318
ERRSTR subroutine 321
ERRTRA subroutine 321
exclamation point 9, 13
executable program
definition 4
names 8
executable statement, definition 4
execution environment, initialization 344
execution error message 379
execution termination 283
execution-time
EXIT subroutine 283
assembler language requirements 350
EXP 246
error message 419
registers used 349
explicit type statement 91
explicitly called routines
assembler information 348
bit function 350
registers used 348
explicitly called subprograms
absolute value 250

explicitly called subprograms *(continued)*
  arcsine and arccosine 248
  arctangent 248
  error function 251
  exponential 246
  gamma and log gamma 251
  hyperbolic sine and cosine 250
  hyperbolic tangent 250
  logarithmic 246
  sine and cosine 248
  square root 251
  tangent and cotangent 249
exponent overflow exception 270
exponent testing 270
exponent, real 17
exponential routines, registers used 348
exponential subprograms
  explicitly called 246
  implicitly called 267
exponentiation 267
expression
  arithmetic 32
  character 37
  evaluation of 31
  examples 32
  kinds of 31
  logical 40
  relational 38
  type of primary in 32
expressions 31, 340
extended error handling 315
extended precision data type 342
extended precision routine results 347
extensions, IBM
  documentation of vi
external
  file
    position 198
    sequential 85
  function name 86
  I/O unit connected to 157
  procedure, definition 4
  unit not connected to 157
EXTERNAL statement 95

F

F format code 102
FALSE 19
FCDCD#
  error message 418
  registers used 349
FCDXI# 265
  error message 418
  registers used 349
FCQCQ#
  error message 423
  registers used 349
FCQXI# 265
  error message 423
  registers used 349
FCXPC#
  error message 417
  registers used 349
FCXPI# 265
  error message 417
  registers used 349
FDXPD# 265
  error message 417
  registers used 349
FDXPI# 265
  error message 417
  registers used 349
field widths needed for data types 230
file
  connection 47
    determining 138
  existence 47
  named 46
  preconnection 47, 138
  status 151
  unnamed 46
file allocation, dynamic
  error messages 437
  with FILEINF subroutine 283
  with INQUIRE statement 137
  with OPEN statement 152
file characteristics 283
file connection 440
  definition 441
  determining
    for an named file 136
    for an unnamed file 138
file deallocation
  error messages 437
file deletion error message 382
file existence
  determining
    for an named file 136
    for an unnamed file 138
file information 283
FILEDEF
  error messages 437
FILEINF subroutine 283
  error messages 379, 410
  examples 287
  rules 286
  with INQUIRE statement 137
  with OPEN statement 152
FIPS flagger 339
fix 53
fixed-form source
  comments 11, 13
  continuation line 11
  example 12
  identification 12
  initial line 11

fixed-form source *(continued)*
label 11
number 207
fixed-length records 5
fixed-view Data-in-Virtual subroutines 294
FIXPI# 265
error message 416
registers used 349
flagger, source language 339
float 54
See also REAL
floating-point
information, saving 346
registers 346
format
with comma as delimiter 99
format codes
begin data transmission (T) 112
blanks, interpretation of (BN) 114
blanks, interpretation of (BZ) 114
character constant transmission (H) 111
character data transmission (A) 110
character skipping (X) 111
comma 99
complex data editing (D,E,Q) 102
decimal point 99
double precision data editing (D,E,Q) 102
end-of-record indication (slash) 115
format control termination (colon) 116
format specification reading 116
group format specification 112
hexadecimal data transmission (Z) 106
integer data editing (I) 101
list-directed formatting 117
logical variable transmission (L) 109
numeric 107
plus character control (S, SP, SS) 113
real data editing (D,E,Q) 102
real data editing (F) 102
real data transmission (G) 103
rules for conversion 97
scale factor specification (P) 104
with comma as delimiter 99
with decimal point 99
format control 98
format identifier (FMT)
READ statement
direct access 166
keyed access 169
sequential access 174
WRITE statement
direct access 218
keyed access 220
sequential access 223
FORMAT statement 96
A code 110
BN code 114
BZ code 114
colon code 116

FORMAT statement *(continued)*
comma 99
D code 102
E code 102
error messages 408
F code 102
forms of 100
G code 103
H code 111
I code 101
L code 109
numeric code 107
output 97
P code 104
printing 97
Q code 102
rules for conversion 97
S code 113
slash code 115
SP code 113
SS code 113
T code 112
X code 111
Z code 106
FORTRAN-supplied procedure
See intrinsic function
FQXPI# 265
error message 418
registers used 349
FQXPQ# 265
error messages 418, 419
registers used 349
FQXPR error messages
exponent out of range 421
invalid argument value 427
FQXP2# 265
error message 421
registers used 349
free-form source
comments 9, 12
continuation line 10
continued line 9
example 10
initial line 9
maximum length 10
minus sign 9
statement label 9, 207
FRXPI# 265
error message 416
registers used 349
FRXPR# 265
error message 417
registers used 349
function
entry name 244
error code 245
evaluating 31
generic name 244
mathematical 244

function *(continued)*
reference 204
value, type, and range 244
FUNCTION statement 120, 204
function subprogram 48
actual arguments 122
dummy arguments 123
END statement 84
ENTRY statement 86
RETURN statement 196

G

G format code 103
gamma and log gamma subprograms 251
gamma routines, registers used 348
GAMMA/ALGAMA 251
error message 427
registers used 349
general rules for data conversion 97
general service subroutines 269
generic functions 341
generic names for intrinsic functions 259
global name 7
shortened form 8
glossary 439
GO TO statement 125
assigned 125
computed 126
unconditional 126
group format
nesting 98
specification 112

H

H format code 111
hexadecimal constant 21
hexadecimal data transmission 106
hierarchy of operations
arithmetic 33
logical 42
Hollerith constant 20
Hollerith constants 341
hyperbolic cosine routines, registers used 348
hyperbolic functions 250
hyperbolic sine and cosine subprograms 250
hyperbolic tangent routines, registers used 348
hyperbolic tangent subprograms 250

I

I format code 101
I/O error messages 402
IABS 250
IAND 257
IBCLR 257
IBCOM error messages
I/O fixup routine 430

IBCOM error messages *(continued)*
secondary entry into main routine 430
IBCOP error message 407
IBM extensions
documentation of vi
IBSET 257
ICHAR function 256
identification, fixed-form 12
identify
statements 207
user-supplied subprogram 95
identifying a VSAM linear data set 296
identifying in Data-in-Virtual 296, 300
IDIM 252
registers used 349
IDNINT 252
IDNINT/IFIX/INT
registers used 349
IEOR 257
IF statement 127
arithmetic 127
block 128
level 128
logical 132
IF-block 128
IFIX
See INT
IMPLICIT statement 133
with double-byte characters 133
implicitly called routines 263
assembler information 349
implicitly called subprograms
exponential 267
implied DO
error message 407
in DATA statement 80
in PRINT statement 81
in READ statement 81
in WRITE statement 81
in-line comment 11
INCLUDE statement 48, 134
examples 136
rules 135
INDEX
error message 404
INDEX function 256
industry standards v
initial line
fixed-form 11
free-form 9
initializing the execution environment 344
input format
fixed-form 4
free-form 4
input records 4
input/output error messages
asynchronous I/O not supported 399
direct 414
direct access mode 399

input/output error messages *(continued)*
  end of record 408
  fixup routine 430
  parallel subroutine 434
  record format invalid 408
  referring to an unopened unit 389
  referring to non-VSAM KSDS file 389
  rewinding the file 426
  sequential 413
  statement processing 390
  subtask abended 407
  unopened VSAM file 400
  VSAM file 400
  WAIT statement 427
input/output semantics 46
input/output statement 46
  BACKSPACE 57
  CLOSE 64
  ENDFILE 85
  FORMAT 96
  implied DO 81
  INQUIRE 136
  OPEN 151
  PRINT 160
  READ 163
  REWIND 198
  underlying semantics 46
  WAIT 214
  WRITE 216
input/output unit
  connected to external file 157
  information, output format 360
  not connected to external file 157
  PRINT statement 81
  READ statement 81
  status information, output format 361
  WRITE statement 81
INQUIRE statement 141
  by file 136
  by unit 138
  by unnamed file 138
  default ddnames 139
  error messages
    FILE specifier 402
  examples 146
  optional specifiers 139, 144
  with double-byte characters 143
  with dynamic allocation 137
INT 253
integer constant 15
integer data editing 101
integer data type 341
INTEGER type statement 91
integer value error message 407
integer variable
  READ statement
    direct access, formatted 166
  storage length 22
integer-to-integer 265
intermediate vector register 354
internal file error message 406
interruption procedures 375
intrinsic function
  definition 4
  explanation 243
  format 243
  generic names for 259
intrinsic functions 341
INTRINSIC statement 147
invalid VS FORTRAN Version 2 programs 3
IOR 257
IOSTAT parameters honored for I/O errors 325
ISHFT 257
ISIGN 252
  registers used 349
ISO definitions 439

K

KEEP 64
key sequenced data set error message 399
keyed access
  INQUIRE statement 141
  READ statement
    formatted 169
    unformatted 189
  WRITE statement
    formatted 220
    unformatted 238
keyed access input/output 341
keywords 13

L

L format code 109
labeled DO statement 77
labeled DO WHILE statement 82
language flagger 339
language reference 1
language syntax 4
LASCN error message 422
LATN2 error message 421
LEN function 256
length of statement 10, 11
length specification
  IMPLICIT statement 133
length, statement 10
lexical compare error message 404
LEXP error message 421
LGAMA error message 429
LGE function 256
LGT function 256
library
  availability 343
  error procedures 375
  interruption procedures 375
  messages 375

library *(continued)*
procedure messages 375
library messages 375
library procedures 375
library reference 261
linear data set, identifying 296
list-directed
formatting 117
PRINT statement 160
READ statement
external 179
internal 181
WRITE statement
external 228
internal 231
list-directed input/output 341
literal constants 341
LLE function 256
LLOG error message 421
LLT function 256
load module processing termination 283
local name 7
log-gamma routines
registers used 349
logarithmic subprograms 246
logical assignment statement 54
logical constant 19
logical data type 341
logical expression 40
order of computations 42
use of parentheses in 43
logical IF statement 132
logical operation, result 44
types 45
logical operator 40
logical primary
See primary
LOGICAL type statement 91
logical variable
storage length 22
transmission 109
LSCN error message 421
LSQRT error message 421
LTNCT error message 422
LXCMP error message 404

M

main program
assign a name to 162
definition 3
PROGRAM statement 162
statement (PROGRAM) 48
mathematical exception tests 270
mathematical library subroutines
explicitly called 348
implicitly called 265
result registers 347
use in assembler language 343
mathematical library subroutines *(continued)*
use in VS FORTRAN Version 2 263
mathematical library subroutines, registers used 348
mathematical library, alternative 263
mathematical routine results 347
mathematical subroutine errors 331
MAX 253
maximum/minimum functions 253
message fragment 359
messages
execution error 379
library 375
operator 376
program interrupt 375
MIN 253
minus sign 6, 9
miscellaneous mathematical functions 250
mixed character constant
definition 19
invalid 30
substring 30
mixed-mode arithmetic expressions 341
MOD 251
registers used 349
modular arithmetic routines, registers used 348
multiply complex numbers 265
Multitasking Facility (MTF) 335
error unit 139, 153
print unit 139, 153
sharing a common block 338

N

named common block 67
named files 46
NAMELIST input data 149
NAMELIST statement 148
error messages 412
examples 150
input data 149
output data 150
READ statement
external 183
internal 186
WRITE statement
external 233
internal 235
names
array 66, 76
block data subprogram 8
block of data 59
CALL statement 86
common-block 8
constant 99
DBCS 8
ddname 46
definition 8
EBCDIC 8
elements of a program 7

names *(continued)*
example 8
external procedure 8
file 46
function reference 86
function subprogram 8
generic 147
global 7
invalid 8
local 7
main program 8
specific 147
subroutine subprogram 8
valid 8
variable 22, 66
naming conventions
default ddnames 139
natural logarithmic routines, registers used 348
new page 98
NINT 252
registers used 349
non-recoverable failure, sample program and output 369
nonexecutable statement 4
nonmathematical arguments 270
NOT 257
NTASKS subroutine 335
numeric character
See arithmetic constant
numeric format code examples 107
numeric input field 118

O

object-time dimensions 28
obtaining a dynamic common length 303
OPEN statement 151
connecting for loading records 155
connecting for reading records 155
connecting for retrieving records 155
connecting for writing records 155
error messages
ACCESS specifier 402
BLANK specifier 403
conflicting specifiers 387
duplicate file name 390
empty file 391
FILE specifier 402
FORM specifier 403
invalid ACTION specifier 392
issued for file already open 387
KEYS specifier 388, 392
no available file 391
status of scratch 403
STATUS specifier 402
examples 156
for connected units 157
for non-connected units 157
with double-byte characters 154

OPEN statement *(continued)*
with dynamic allocation 152
operator messages 376
operators, arithmetic 32
option default 325
option table
default values 325
entry 322
error 315
order of statements in a program unit 49
output format
array 358
control flow information 359
I/O unit information 360
I/O Unit Status Information 361
variable character 358
variable noncharacter 357
output from symbolic dumps 357
OVERFL subroutine 270
overflow
error message 377
indicator service routine 270

P

P format code 104
packets, debug 72
page control 98
parallel subroutine
DSPTCH subroutine 336
NTASKS subroutine 335
SHRCOM subroutine 338
SYNCRO subroutine 337
PARAMETER statement 158
parameters, correct 346
parentheses
error message 398
in arithmetic expressions 34
use in character expressions 38
use in logical expressions 43
PAUSE statement 159
error message 376
period v, 6
plus sign 6
preconnected file
definition 4
OPEN statement 47
WRITE statement
formatted with direct access 219
formatted with sequential access 224
list-directed with external devices 229
NAMELIST with external devices 234
unformatted with direct access 237
unformatted with sequential access 241
preserving shift codes 277
primary 32
logical 40
PRINT statement 160
implied DO in 81

- printouts, sample storage 357
- procedure
  - definition 3
  - external 4
- processing termination 283
- program
  - basic elements of 3
  - executable 4
  - main 3
- program interrupt error 378
- program mask 353
- PROGRAM statement 48, 162
- program unit
  - definition 3
  - order of statements in 49
- program-interrupt messages 330, 375

Q

- Q format code 102
- QABS 250
  - registers used 349
- QARCOS
  - See QARSIN/QARCOS
- QARSIN/QARCOS 248
  - error message 428
  - registers used 349
- QASCN error message 428
- QATAN/QATAN2 248
  - error message 428
  - registers used 349
- QCONJG 251
  - registers used 348
- QDIM 252
  - registers used 349
- QERF/QERFC 251
  - registers used 349
- QEXP 246
  - error message 427
  - registers used 349
- QEXT 253
- QIMAG 251
  - registers used 349
- QINT 251
  - registers used 349
- QLOG/QLOG10 246
  - error message 428
  - registers used 349
- QMOD 251
  - registers used 349
- QP
  - extend 54
  - float 54
- QSCN error message 428
- QSCNH error message 428
- QSIGN 252
  - registers used 349
- QSIN/QCOS 248
  - error message 428
- QSIN/QCOS *(continued)*
  - registers used 349
- QSINH/QCOSH 250
  - error message 428
  - registers used 349
- QSQRT 251
  - error message 427
  - registers used 349
- QTAN/QCOTAN 249
  - error message 429
  - registers used 349
- QTANH 250
  - registers used 349
- QTNCT error message 429
- quotation mark 6, 9

R

- READ statement
  - asynchronous 163
  - error messages 388
    - file not created 414
    - implied DO 407
    - KEY argument 388
  - formatted with
    - direct access 166
    - keyed access 169
    - sequential access 173
    - sequential access, internal 176
  - forms of 163
  - implied DO in 81
  - list-directed I/O with
    - external devices 179
    - internal files 181
  - NAMELIST with
    - external devices 183
    - internal files 186
  - unformatted with
    - direct access 187
    - keyed access 189
    - sequential access 193
- REAL 253
- real assign 53
- real constant 16
  - examples 17
- real data of length 8 23
- real exponent 17
- REAL type statement 91
- real variable, storage length 22
- real-to-integer 265
- real-to-real 265
- record length error message 399
- record number error message 414
- record, fixed length 5
- record, variable length 5
- relational expression 38, 340
- relational operator 38
- reposition a file 57

- request traceback 321
- resetting in Data-in-Virtual 305
- return code subroutines
  - SYSRCS 276
  - SYSRCT 276
  - SYSRCX 276
- RETURN statement
  - in a function subprogram 196
  - in a subroutine subprogram 196
- REWIND statement 198
- REWRITE statement
  - formatted 199
  - unformatted 201
- rewriting records 199
- routines
  - explicitly called 263
  - implicitly called
    - character 266
    - mathematical 265
    - service 267
- rules
  - arithmetic expression construction 32
  - data conversion 97
  - language 3
- run-time termination 283

S

- S format code 113
- SAA flagger 339
- sample programs and output
  - array items 364
  - non-recoverable failure 369
  - variable items 362
- sample storage printout 357
- SASCN error message 420
- SATN2 error message 420
- save option table entry 317
- SAVE statement 203
- saving changes in Data-in-Virtual 305
- scale factor 104
- SCNH error message 422
- SDUMP subroutine
  - description 274
  - format specifications 274
  - message fragment 359
  - output 274
  - output format 357
  - output of symbolic dump 361
  - programming considerations 274
  - sample printouts 357
- sequential access
  - INQUIRE statement 141
  - READ statement
    - formatted 173
    - unformatted 193
  - WRITE statement
    - formatted 223
    - unformatted 240
- sequential access input/output 342
- service routines
  - assembler information 345
  - ASSIGNM 277
  - CDUMP/CPDUMP 273
  - CLOCK 278
  - CPUTIME 280
  - Data-in-Virtual subroutines 293
    - DIVCML 303
    - DIVINF 296
    - DIVINV 300
    - DIVRES 305
    - DIVSAV 305
    - DIVTRF 299
    - DIVTRV 304
    - DIVVWF 298
    - DIVVWV 302
  - DATIM 282
  - DUMP/PDUMP 271
  - end processing 283
  - error-handling subroutines 315
    - ERRMON subroutine 316
    - ERRSAV subroutine 317
    - ERRSET subroutine 318
    - ERRSTR subroutine 321
    - ERRTRA subroutine 321
  - EXIT 283
  - FILEINF 283
  - mathematical exception test 270
  - Multitasking Facility (MTF) Subroutines 335
    - DSPTCH 336
    - NTASKS 335
    - SHRCOM 338
    - SYNCRO 337
  - return code subroutines 276
    - SYSRCS 276
    - SYSRCT 276
    - SYSRCX 276
  - SDUMP 274
  - sizes 345
  - storage dump 271
  - SYSABD 287
  - SYSABN 287
  - UNTANY 288
  - UNTNOFD 288
  - XUFLOW 291
- service subprograms
  - assembler information 350
- service subroutines 269
- SEXP error message 419
- SGAMA error message 427
- shareable load 395, 396
- sharing a common block 338
- shift-out/shift-in characters
  - definition 444
  - documentation v
  - I/O support 154
  - in character set 5
  - in continued source lines 10, 12

shift-out/shift-in characters *(continued)*
  in source 11
SHRCOM subroutine 338
  error messages 398
SIGN 252
  registers used 349
SIN 248
  error message 419
  registers used 349
SIN/COS error message 419
sine and cosine subprograms 248
sine routines, registers used 348
SINH/COSH 250
  error message 420
  registers used 349
size and type declaration of an array 26
skip a line 98
slash 6
slash format code 115
SLOG error message 419
source language flagger 339
source language statement
  fixed-form 11
  free-form 9
source statement characters
  digit 6
  letter 6
  special characters 6
SP format code 113
space considerations 347
special characters 6
specification statement 48
  CHARACTER type 91
  COMMON 66
  COMPLEX type 91
  DIMENSION 76
  DOUBLE PRECISION type 91
  EQUIVALENCE 89
  explicit type 91
  EXTERNAL 95
  IMPLICIT 133
  INTEGER type 91
  INTRINSIC 147
  LOGICAL type 91
  NAMELIST 148
  PARAMETER 158
  REAL type 91
  SAVE 203
SQRT 251
  error message 419
  registers used 349
square root routines
  example with vector registers 355
  examples 351
  registers used 348
square root subprograms 251
SS format code 113
SSCN error message 419
SSCNH error message 420
SSQRT error message 419
statement
  categories 45
  descriptions 50
  executable 4
  explicit 23
  fixed-form number 11, 207
  free-form label 207
  functions of 45
  implicit 23
  inserting with INCLUDE 134
  label 13, 207
    fixed-form 11
    free-form 9
  maximum length 10
  nonexecutable 4
  number 51, 207
  specification 23
statement function 342
statement function statement 204
statement label
  READ statement
    direct access, formatted 166
statement length 10, 11
statement order 49
status of a file 151
STNCT error message 420
stop display 213
stop program 83
STOP statement 207
  error message 376
storage dump 271
  See also utility routines
storage dump service routines 271
storage estimates, extended precision routines 345
storage printout, sample 357
storage, shared 66, 89
store entry in option table 321
SUBCHK function of DEBUG 72
subprogram
  BLOCK DATA statement 48, 59
  calling in assembler 345
  definition 3
  ENTRY statement 86
  FUNCTION statement 48, 120
  initialization instructions 344
  SUBROUTINE statement 48, 208
subprogram statements 48
subprograms, explicitly called 263
  absolute value 250
  arcsine and arccosine 248
  arctangent 248
  bit manipulation 257
  character manipulation 256
  conversion 253
  error function 251
  exponential 246
  gamma and log gamma 251

subprograms, explicitly called *(continued)*
hyperbolic sine and cosine 250
hyperbolic tangent 250
logarithmic 246
maximum/minimum 253
sine and cosine 248
square root 251
tangent and cotangent 249
subprograms, implicitly called
exponential 267
SUBROUTINE statement 208
subroutine subprogram
actual arguments 209
dummy arguments 209
END statement 84
ENTRY statement 86
RETURN statement 196
service and utility subroutines 269
subroutines for programming tasks 269
subroutines, general service 269
subscript
definition 25
examples 25
identify array element 25
in DATA statement 69
subscripts 25
substring
character 28
expression 28
in DATA statement 69
symbolic dump
array items 364
examples 361
how to call 274
non-recoverable failure 369
output format 357
variable items 361
symbolic dumps, output 357
symbolic name
See names
SYNCRO subroutine 337
syntax 4
SYSABD subroutine 287
SYSABN subroutine 287
SYSRCS subroutine 276
SYSRCT subroutine 276
SYSRCX subroutine 276
Systems Application Architecture flagger 339

T

T format code 112
table, error option 315
TAN 249
registers used 349
TAN/COTAN error message 420
tangent and cotangent subprograms 249
tangent routines
registers used 349
TANH 250
registers used 349
task subroutines 269
terminate processing (EXIT) 283
terminating access in Data-in-Virtual 299, 304
test
exponents 270
for divide-check exception 270
time subroutines
CLOCK and CLOCKX 278, 282
DATIM 282
DATIMX 282
TRACE OFF statement 213
in debug packet 72
TRACE ON statement 213
in debug packet 72
traceback 369
traceback request 321
trigonometric functions 248
trigonometric subprograms, error message 419
TRUE 19
truncation routines
registers used 349
two-to-real 265
type declaration
DBCS names 24
declaration of an array 26
EXPLICIT statement 24
IMPLICIT statement 24
predefined 23
type declaration by predefined specifications 23
type specification 133

U

unary signs 33
unconditional GO TO statement 126
underflow error message 378
underscore 6
unformatted I/O error message 408
unit
connected 138
connected to external file 157
INQUIRE statement 138
not connected to external file 157
number 138, 151
OPEN statement 151
unit connection 440
unit connection, determining 138
UNIT specifier
error messages 402
on the INQUIRE statement 138, 139
unlabeled DO statement 77
unlabeled DO WHILE statement 82
unnamed files 46
UNTANY subroutine 288
UNTNOFD subroutine 288
updating records 199

user exit routine, coding the 319
utility and service subroutines
  CDUMP/CPDUMP 273
  DUMP/PDUMP 271
  EXIT 283
  SDUMP 274
  XUFLOW 291

V

valid VS FORTRAN Version 2 programs 3
value separator 118
VAMTP error message 434
variable
  character 28
  declaration 23
  definition 22
  lengths 22
  names 22
  types 22
variable character, output format 358
variable items, sample program and output 362
variable noncharacter, output format 357
variable-length records 5
varying-view Data-in-Virtual subroutines 294
VASYN error messages
  unit number out of range 411
VASYP error messages
  asynchronous I/O ddname 404, 405
  asynchronous I/O not supported 399
  asynchronous I/O subtask 407
  blocksize not specified 415
  end of data set 409
  end of record 408
  I/O error 409
  invalid address 413
  no wait issued for outstanding I/O request 427
  REWIND statement 426
  unformatted I/O 408
  WAIT statement 427
VCIA4 error message 406
VCLOP error message 403
VCOMH error messages
  end of record 408
  invalid character in FORMAT statement 408
  nested parentheses overflow 398
VCOM2 error messages
  address conflict in link mode 393
  composite module conflict 393
  I/O fixup routine 430
  module loaded not recognized 394
  secondary entry into main routine 430
  specifier list conflict 396
VCTVH error messages
  illegal decimal character 409
  illegal hexadecimal character 412
  integer value out of range 407
  real value out of range 412
VDIOS error messages
  direct access data set 414
  direct access I/O 413
  direct access read 414
  direct/keyed access I/O statement 434
  FORMATTED and UNFORMATTED I/O requests 401
  I/O error 409
  incorrect record length specified 415
  OPEN request ignored 401
  OPEN statement ACTION specifier conflict 387
  record length 414
  record number 414
  record number and length 396
  SCRATCH file 401
  unit number out of range 411
VDIVP error messages 394
vector
  count register 353
  error handling 353
  interruption index 353
  intrinsic function
    calling sequence 351
    result registers 347
  mask register 353
  registers, passing arguments in 351
vector entry points 354
vector register, intermediate 354
VEMGN error message 430
  error count 429
VEMGN error messages
  error monitor 430
  text for error message 431
VFNTH error messages
  divide by zero 378
  overflow 377
  program interruption 378
  underflow 378
VIIOS error message 406
VINQP error message 402
VINQP error messages 410
VIOCP error message 430
VIOFP error message 430
VIOLP error messages
  end of record 408
  error in repeat count 413
  incorrect delimiter 415
  item size exceeds buffer length 407
VIONP error messages
  end of record 412
  invalid subscript 412
  NAMELIST statement 412
VIOUF/VIOFM error message 389
VIOUP error messages 406
  end of record 408
  I/O fixup routine 430
VKIOS error messages
  add record with duplicate key 391
  direct/keyed access I/O statement 434

VKIOS error messages *(continued)*
file name 390
I/O statement processing 390
I/O statement refers to non-VSAM KSDS file 389
inconsistent record lengths 390
key argument length too long 388
key of reference conflict 392
key specifies no available file 391
KEYID specifier conflict 388
loss of file position 387
open an empty file 392
OPEN statement ACTION specifier conflict 387
OPEN statement with conflictinq specifiers 387
record loading 393
record not found with specified key 388
record too short 389
VLCIO error messages
abend 431
library initialization 436
task initialization 435
VLINP error messages
load module does not contain shareable part 395
load module has incorrect format 395
part timestamp conflict 396
unable to branch to nonshareable part 395
VLOAD error message 396
VMEXP error messages
addressing mode conflict 435
multitasking facility 435
multitasking facility function 433
parallel subroutine load module 433
VMINI error messages
main task/load module conflict 435
parallel subroutine load module 432
VMMAA error messages
autotask dd statement 432
multitasking facility subtask 434
parallel subroutine load module 432
VMNTP error messages
multitasking facility function invalid 433
multitasking facility function without arguments 433
VMOPP error messages
error number out of range 404
unmodifiable message table entry 405
VMPRM error message 432
VMSYP error message 433
VMTRM error messages
multitasking facility 435
multitasking facility subtask 434
parallel subroutine 435
VOPEP error messages 410
ACTION specifier 392
invalid access specifier 402
invalid blank specifier 403
invalid file specifier 402
invalid form specifier 403
invalid status specifier 402
KEYS specifier conflict 392

VOPEP error messages *(continued)*
OPEN statement issued for file already open 387
recl specifier conflict 397
status of scratch not allowed 403
VPARM error messages
autotask keyword 413
autotask keyword subparameter 432
invalid autotask keyword 431
VSAM error messages 399
VSAM linear data set, identifying 296
VSCOM error messages
I/O fixup routine 430
secondary entry into main routine 430
VSERH error message 413
VSIOS error messages
CLOSE statement 401
direct I/O 414
end of data set 409
FORMATTED and UNFORMATTED I/O requests 401
I/O error 409
I/O in a parallel subroutine 434
OPEN request ignored 401
OPEN statement ACTION specifier conflict 387
REWIND statement 426
SCRATCH file 401
sequential I/O 413
unformatted I/O 408
unit number out of range 411
VSIOS/VDIOS error message 396
VSPAP error message 403
VSPIP error message 436
VSTAE error message 415
VSTIO error messages 437
VVIOS error messages 409
ENDFILE statement 400
file positioning I/O statement in direct access mode 399
I/O request made to unopened VSAM file 400
I/O statement 400
key sequenced data set 399
OPEN statement ACTION specifier conflict 387
record addition conflict 399
record length too long 399
record number 414
unit number out of range 411

W

WAIT statement 214
error messages 427
valid 215
write an end-of-file record 85
WRITE statement 216
asynchronous 216
error messages 399, 407
formatted with
direct access 218
keyed access 220
sequential access 223

WRITE statement *(continued)*
- formatted with *(continued)*
  - sequential access, internal 225
- forms of 216
- implied DO in 81
- list-directed I/O with
  - external devices 228
  - internal files 231
- NAMELIST with
  - external devices 233
  - internal files 235
- unformatted with
  - direct access 236
  - keyed access 238
  - sequential access 240

X

X format code 111
XUFLOW subroutine 291

Z

Z format code 21, 106

VS FORTRAN Version 2
Language and Library
Reference

SC26-4221-3

Reader's Comment Form

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: ______________________

Chapter/Section ______________________

______________________ Page No. ______________________

Comments:

If you want a reply, please complete the following information.

Name ______________________ Phone No. (____) ______________________

Company ______________________

Address ______________________

____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

SC26-4221-3

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023

Fold and tape | Please do not staple | Fold and tape

IBM®

VS FORTRAN Version 2
Language and Library
Reference

Reader's Comment Form

SC26-4221-3

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: ____________________

Chapter/Section ____________________

____________________ Page No. ____________________

Comments:

If you want a reply, please complete the following information.

Name ____________________ Phone No. (______) ____________________

Company ____________________

Address ____________________

____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

SC26-4221-3

Reader's Comment Form

Fold and tape   Please do not staple   Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL   PERMIT NO. 40   ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023

Fold and tape   Please do not staple   Fold and tape

IBM®

VS FORTRAN Version 2
Language and Library
Reference

Reader's Comment Form

SC26-4221-3

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: ____________________

Chapter/Section ____________________

____________________ Page No. ____________________

Comments:

If you want a reply, please complete the following information.

Name ____________________ Phone No. (______) ____________________

Company ____________________

Address ____________________

____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

SC26-4221-3

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023

Fold and tape | Please do not staple | Fold and tape

IBM ®

Program Number
5668-805
5668-806

File Number
S370-40

The VS FORTRAN Version 2 Library

Diagnosis Guide	LY27-9516
General Information	GC26-4219
Installation and Customization for MVS	SC26-4340
Installation and Customization for VM	SC26-4339
Interactive Debug Guide and Reference	SC26-4223
Language and Library Reference	SC26-4221
Licensed Program Specifications	GC26-4225
Programming Guide	SC26-4222
Reference Summary	SX26-3751

SC26-4221-3

Printed in U.S.A.