

VS FORTRAN Version 2



Language and Library Reference

Release 6

VS FORTRAN Version 2



Language and Library Reference

Release 6

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page x.

ReadMe!

Additional information, made available after publication of this edition, may be found in the VS FORTRAN Version 2 README file. This file is updated with new information pertaining to this product as the result of service updates, as well as any additional information provided in response to Reader Comment Forms. On CMS, this file name is named VSF2 README and is located on the disk on which the VS FORTRAN Version 2 product is installed. On MVS, this file is named SYS1.VSF2.README (note: your product installer may have chosen a name other than those given here; check with your local product support for the correct name used by your installation). You may browse or print this file using local procedures to view this information.

Ninth Edition (November 1993)

This edition replaces and makes obsolete the previous editions, SC26-4221-04, SC26-4221-05, SC26-4221-06, and SC26-4221-07.

This edition applies to VS FORTRAN Version 2 Release 6, Program Numbers 5668-805, 5688-087, and 5668-806, 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 "About This Book." Specific changes for this edition are indicated by a vertical bar to the left of the change. A vertical bar to the left of a figure caption indicates that the figure has changed. 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.

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 Reader's Comment Form is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Canada Ltd. Laboratory Information Development 2G/345/1150/TOR 1150 Eglinton Avenue East North York, Ontario, Canada M3C 1H7

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1986, 1993. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	x
Programming Interface Information	x
Trademarks and Service Marks	x
 About This Book	xi
How This Book Is Organized	xi
How to Use This Book	xii
Syntax Notation	xii
Publications	xiii
Industry Standards	xv
Documentation of IBM Extensions	xv
Operating System Support	xvi
 Summary of Changes	xvii
Release 6, October 1993	xvii
Major Changes to the Product	xvii
Release 5, September 1991	xix
Major Changes to the Product	xix
Release 4, August 1989	xx
Major Changes to the Product	xx
Release 3, March 1988	xxi
Major Changes to the Product	xxi
Release 2, June 1987	xxii
Major Changes to the Product	xxii
Release 1.1, September 1986	xxiii
Major Changes to the Product	xxiii

Part 1. Language Reference 1

Chapter 1. Language	2
Valid and Invalid Programs	2
Language Definitions	2
Language Syntax	4
Input Records	4
Source Statement Characters	4
Names	7
Source Language Statements	9
Statement Labels	12
Keywords	13
 Chapter 2. Data	14
Constants	14
Arithmetic Constants	14
Logical Constants	18
Character Constants	19
Hollerith Constants	19
Binary Constants	20
Octal Constants	20
Hexadecimal Constants	21

Using Typeless Constants	22
Variables	23
Variable Names	23
Data Types and Lengths	23
Arrays	26
Subscripts	26
Size and Type Declaration of an Array	27
Character Substrings	30
 Chapter 3. Expressions	32
Evaluation of Expressions	32
Arithmetic Expressions	33
Arithmetic Operators	33
Rules for Constructing Arithmetic Expressions	33
Use of Parentheses in Arithmetic Expressions	35
Type and Length of the Result of Arithmetic Expressions	35
Examples of Arithmetic Expressions	37
Character Expressions	38
Concatenation Operation in Character Expressions	38
Use of Parentheses in Character Expressions	39
Constant Expressions	39
Relational Expressions	39
Logical Expressions	41
Logical Operators	41
Order of Computations in Logical Expressions	43
Use of Parentheses in Logical Expressions	45
Using Context-Specific Data Types in Expressions	46
Using Typeless Constants	46
Using BYTE Data Objects	46
 Chapter 4. Statements	47
Statement Categories	47
Assignment Statements	47
Control Statements	47
Parallel Statements	47
DATA Statement	48
Debug Statements	48
Input/Output Statements	48
PROGRAM Statement	50
Specification Statements	50
Subprogram Statements	50
Compiler Directives	51
Allocation Statements	51
Order of Statements in a Program Unit	51
Statement Descriptions	52
@PROCESS Statement	53
ALLOCATE Statement	54
Arithmetic IF Statement	55
ASSIGN Statement	55
Assigned GO TO Statement	56
Assignment Statements	57
AT Statement	62
AUTOMATIC Statement	63
BACKSPACE Statement	63

BLOCK DATA Statement	65
Block IF Statement	67
CALL Statement	67
CHARACTER Type Statement	71
CLOSE Statement	71
COMMON Statement	73
COMPLEX Type Statement	76
Computed GO TO Statement	76
CONTINUE Statement	76
DATA Statement	76
DEALLOCATE Statement	78
DEBUG Statement	79
DELETE Statement	84
DIMENSION Statement	85
DISPLAY Statement	86
DO and DO WHILE Statements	86
DOUBLE COMPLEX Type Statement	91
DOUBLE PRECISION Type Statement	91
EJECT Statement	91
ELSE Statement	91
ELSE IF Statement	91
END Statement	91
END DEBUG Statement	92
END DO Statement	92
ENDFILE Statement	92
END IF Statement	94
ENTRY Statement	94
EQUIVALENCE Statement	97
Explicit Type Statement	99
EXTERNAL Statement	103
FORMAT Statement	104
FUNCTION Statement	131
GO TO Statements	135
IF Statements	138
IMPLICIT Statement	143
INCLUDE Statement Directive	145
INQUIRE Statement	147
INTEGER Type Statement	163
INTRINSIC Statement	163
Logical IF Statement	164
LOGICAL Type Statement	164
NAMelist Statement	164
NULLIFY Statement	167
OPEN Statement	168
PARAMETER Statement	176
PAUSE Statement	178
POINTER Statement	178
PRINT Statements	180
PROGRAM Statement	182
READ Statements	183
REAL Type Statement	217
RETURN Statement	217
REWIND Statement	219
REWRITE Statement—Formatted with Keyed Access	220

REWRITE Statement—Unformatted with Keyed Access	222
SAVE Statement	224
Statement Function Statement	225
Statement Labels	228
STATIC Statement	228
STOP Statement	229
SUBROUTINE Statement	229
TRACE OFF Statement	233
TRACE ON Statement	234
Unconditional GO TO	234
WAIT Statement	234
WRITE Statements	236
 Chapter 5. Parallel Language Constructs	 264
Parallel Task Management Statements	264
ORIGINATE Statements	264
SCHEDULE Statements	265
WAIT FOR Statements	267
TERMINATE Statement	268
Parallel Loop Statements	269
PARALLEL DO Statement	269
LOCAL Statement (for Parallel Loops)	270
DOBEFORE Statement	271
DOEVERY Statement	272
DOAFTER Statement	272
EXIT Statement	273
Parallel Sections Statements	274
PARALLEL SECTIONS Statement	274
LOCAL Statement (for Parallel Sections)	275
SECTION Statement	275
END SECTIONS Statement	276
Parallel Call Statements	277
PARALLEL CALL Statement	277
WAIT FOR ALL CALLS Statement	278
 Chapter 6. Intrinsic Functions	 280
Intrinsic Function Table	280
Math Function Accuracy Information	292
Units in the Last Place (ULPs)	292
Control of Program Exceptions in Mathematical Functions	293
Argument and Accuracy Information	295

Part 2. Library Reference 309

Chapter 7. Mathematical, Character, and Bit Routines	310
Explicitly Called Routines	310
Implicitly Called Routines	310
Error Conditions for Implicitly Called Functions	315
Complex Divide and Multiply Functions	315
Exponentiation of a Real Base to an Integer Power	316
Exponentiation of a Base Argument to a Real Power	316
Exponentiation of a Real Base to a Real Power	316
Complex Exponentiation	316

Exponentiation of a Complex Base to an Integer Power	316
Exponentiation of a Complex Base to a Complex Power	317
Alternative Mathematical Library Subroutines	317
Chapter 8. Service Subroutines	318
Mathematical Exception Routines	319
DVCHK Subroutine	319
OVERFL Subroutine	319
XUFLOW Subroutine	320
Storage Dump Subroutines	321
DUMP/PDUMP Subroutines	321
CDUMP/CPDUMP Subroutines	322
SDUMP Subroutine	323
Return Code Subroutines	324
SYSRCS Subroutine	324
SYSRCT Subroutine	325
SYSRCX Subroutine	325
Termination Subroutines	326
EXIT Subroutine	326
SYSABN/SYSABD Subroutines	326
File Service Subroutines	327
FILEINF Subroutine	327
UNTNOFD/UNTANY Subroutines	331
IGNFHU/IGNFHDD Subroutines	334
Date/Time Subroutines	336
CLOCK/CLOCKX Subroutines	336
CPUTIME Subroutine	337
DATIM/DATIMX Subroutines	339
Other Service Subroutines	340
ASSIGNM Subroutine	340
ARGSTR Subroutine	342
MVBITS Subroutine	343
Chapter 9. Routines for Use with Parallel Programs	344
Parallel Event Services	345
PEORIG Routine	346
PEPOST Routine	348
PEWAIT Routine	348
PETERM Routine	348
Parallel Lock Services	349
PLORIG Routine	351
PLLOCK Routine	352
PLCOND Function	353
PLFREE Routine	354
PLTERM Routine	355
Virtual Processor Query	356
NPROCS Function	356
Parallel Execution Control	357
PFAFFS Routine	357
PFAFFC Routine	357
PYIELD Routine	358
Parallel Trace Facility	358
PTPARM Routine	359
PTWRIT Routine	359

Chapter 10. Data-in-Virtual Subroutines	361
Data-in-Virtual Overview	361
Using Data-in-Virtual in an MTF Environment	362
Using Data-in-Virtual in a Parallel Program	362
Fixed-View Versus Varying-View Subroutines	362
Syntax of the Subroutines	364
Syntax of Fixed-View Subroutines	364
Syntax of Varying-View Subroutines	368
Syntax of Common Subroutines	372
Interface to the Data-in-Virtual Functions	374
Sample Program with Fixed-View Subroutines	374
Sample Program with Varying-View Subroutines	376
Remapping a Common to Different Parts of the Data Object	380
Resetting a Common	381
Ensuring Data Integrity	381
Performance and Storage Factors	381
 Chapter 11. Extended Error-Handling Topics	 382
Extended Error Handling	382
Error-Handling Subroutines	383
ERRMON Subroutine	383
ERRSAV Subroutine	384
ERRSET Subroutine	385
ERRSTR Subroutine	388
ERRTRA Subroutine	388
Error Option Table	389
 Chapter 12. Multitasking Facility (MTF) Subroutines	 401
NTASKS Subroutine	401
DSPTCH Subroutine	402
SYNCRO Subroutine	403
SHRCOM Subroutine	403
 Appendix A. Summary of the VS FORTRAN Version 2 Language	 405
Source Language Flaggers	405
Major Elements of the VS FORTRAN Version 2 Language	405
FORTRAN 66 Features Not in FORTRAN 77	410
 Appendix B. Assembler Language Information	 412
Library Availability	412
Initializing the Execution Environment	413
Calling Sequences	413
Assembler Language Calling Sequence	413
Entry Names for VS FORTRAN Routines and Functions	416
Explicitly Called Routines	416
Implicitly Called Mathematical Routines	416
Explicitly Called Character Routines	416
Implicitly Called Character Routines	417
Service Routines	417
Explicitly Called Bit Functions	417
Mathematical Routine Results	417
Calling Vector Intrinsic Elementary Functions	417
Required Contents of Registers	417
Program Mask	419

Error Handling	419
Examples of Assembler Language Calling Sequence	419
Entry Names for Vector Intrinsic Elementary Functions	420
Appendix C. Sample Storage Printouts	423
Output from Symbolic Dumps	423
Output Format	423
Variable Noncharacter	423
Variable Character	424
Array	425
Control Flow Information	426
I/O Unit Information	426
I/O Unit Status Information	427
Examples of Sample Programs and Symbolic Dump Output	427
Appendix D. Library Procedures and Messages	438
Library Interruption Procedures	438
Library Error Procedures	438
Library Messages	438
Operator Messages	439
Program-Interrupt Messages	440
Execution Error Messages	441
Index	505

Notices

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 product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectable rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Programming Interface Information

This book is intended to help you create application programs using the VS FORTRAN Version 2 licensed program. This book documents General-Use Programming Interface and Associated Guidance Information provided by VS FORTRAN Version 2.

General-Use programming interfaces allow the customer to write programs that obtain the services of VS FORTRAN Version 2.

VS FORTRAN Version 2 provides no macros that allow a customer installation to write programs that use the services of VS FORTRAN Version 2.

Warning: Do not use as programming interfaces any VS FORTRAN Version 2 macros.

Trademarks and Service Marks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

AIX
AIX/ESA
ESA/390
IBM
MVS/ESA
MVS/SP
MVS/XA
RACF
SAA
Systems Application Architecture
VM/ESA
VM/XA

About This Book

How This Book Is Organized

This book is organized as follows:

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 can use VS FORTRAN Version 2.

Chapter 3, “Expressions,” explains the five kinds of expressions: arithmetic, character, constant, relational, and logical.

Chapter 4, “Statements,” presents the syntax rules and conventions of the VS FORTRAN Version 2 language statements.

Chapter 5, “Parallel Language Constructs,” presents the syntax rules and conventions for the Parallel Language Constructs.

Chapter 6, “Intrinsic Functions,” contains the usage information for the explicitly-called routines commonly used for mathematical computations and character and bit conversions.

Chapter 7, “Mathematical, Character, and Bit Routines,” contains usage information about the implicitly-called routines commonly used for computations and conversions.

Chapter 8, “Service Subroutines,” provides you with usage information concerning the subroutines that have been provided for general programming tasks.

Chapter 9, “Routines for Use with Parallel Programs,” provides information regarding parallel processing of programs running under CMS and MVS.

Chapter 10, “Data-in-Virtual Subroutines,” presents information on the subroutines that allow you to use the data-in-virtual facility on MVS/XA* or MVS/ESA*.

Chapter 11, “Extended Error-Handling Topics,” gives you information about the subroutines that are used for error handling.

Chapter 12, “Multitasking Facility (MTF) Subroutines,” presents information on the subroutines that allow you to use the multitasking facility when running under MVS/XA or MVS/ESA.

Appendix A, “Summary of the VS FORTRAN Version 2 Language,” contains a summary of VS FORTRAN Version 2 language, identifies which are flagged when either the FIPS compiler or the SAA* compile-time option is specified, and contains a list of the FORTRAN 66 features.

* MVS/ESA, MVS/XA, and SAA are trademarks of the International Business Machines Corporation.

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 unrecoverable failure.

Appendix D, “Library Procedures and Messages,” contains the explanations of the program-interrupt and error procedures used by the *VS FORTRAN Version 2 Language and Library Reference*.

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*. 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 list of items explains how to interpret the syntax used in this manual.

Special Note: In the on-line version of this manual, there are a number of symbols which are not supported (displayable). This may make some of the formulae confusing. If any text is not clear, you are referred to the hardcopy of this manual for clarification. In particular greek letters, mathematical symbols, exponents, subscripts, and square brackets are some of the most widely used characters which cannot be displayed. In all cases, we have endeavored to use intuitive representations, but these may vary from one instance to another. The hardcopy will always serve as the best reference.

- Uppercase letters and special characters (such as commas and parentheses) are to be coded exactly as shown, except where otherwise noted. You can 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. Since square brackets are not displayable, we will usually represent them as double triangular brackets.
- OR signs (|) indicate you may choose only one of the items they separate.

* IBM is a trademark of the International Business Machines Corporation.

- Blanks in Fortran statements are used to improve readability; they have no significance, except when shown within apostrophes (' ') or quotation marks (" "). In non-Fortran statements, blanks may be significant. Code non-Fortran statements exactly as shown.




For example, given the following 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 left half of a double-byte EBCDIC character (X'42')
- kk represents a double-byte character not in the double-byte EBCDIC character set

Publications

Figure 1 lists the VS FORTRAN Version 2 publications and the tasks they support.

Figure 1. VS FORTRAN Version 2 Publication Library

Task	VS FORTRAN Version 2 Publication	Order Number
Evaluation and Planning	<i>General Information</i>	GC26-4219
	<i>Licensed Program Specifications</i>	GC26-4225
Installation and Customization	<i>Installation and Customization for CMS</i>	SC26-4339
	<i>Installation and Customization for MVS</i>	SC26-4340
Application Programming	<i>Language and Library Reference</i>	SC26-4221
	<i>Programming Guide for CMS and MVS</i>	SC26-4222
	<i>Interactive Debug Guide and Reference</i>	SC26-4223
	<i>Reference Summary</i>	SX26-3751
Library Reference	<i>Master Index and Glossary</i>	SC26-4603
Diagnosis	<i>Diagnosis Guide</i>	LY27-9516
Migration	<i>Migration from the Parallel FORTRAN PRPQ</i>	SC26-4686

Figure 2 lists additional publications that you might need while using VS FORTRAN.

Figure 2. Related Publications

Title	Order Number
<i>Access Method Services for the Integrated Catalog Facility</i>	SC26-4562
<i>American National Dictionary for Information Processing</i>	—
<i>IBM System/370 Extended Architecture Principles of Operation</i>	SA22-7085
<i>IBM System/370 Principles of Operation</i>	GA22-7000
<i>Interactive Storage Management Facility User's Guide</i>	SC26-4563
<i>An Introduction to Data-in-Virtual</i>	GG66-0259
<i>MVS/370 Data Administration: Macro Instruction Reference</i>	GC26-4507
<i>MVS/370 Message Library: System Codes</i>	GC38-1008
<i>MVS/ESA Application Development Guide</i>	GC28-1672
<i>MVS/ESA System Codes</i>	GC28-1815
<i>MVS/ESA System—Data Administration</i>	SC26-4515
<i>MVS/ESA VSAM Administration: Macro Instruction Reference</i>	SC26-4517
<i>MVS/XA Data Administration: Macro Instruction Reference</i>	GC26-4141
<i>MVS/XA Message Library: System Codes</i>	GC28-1157
<i>MVS/XA Supervisor Services and Macro Instructions</i>	GC28-1154
<i>MVS/XA System Programmer's Library: System Macros and Facilities, Volume 1</i>	GC28-1150
<i>MVS/XA VSAM Administration: Macro Instruction Reference (for DFP Version 1)</i>	GC26-4016
<i>MVS/XA VSAM Administration: Macro Instruction Reference (for DFP Version 2)</i>	GC26-4152
<i>OS/VS Message Library: VS2 System Codes</i>	GC28-1008
<i>OS/VS Virtual Storage Access Method (VSAM) Programmer's Guide</i>	GC26-3838
<i>MVS/DFP Macro Instructions for Data Sets</i>	SC26-4747
<i>IBM High Level Assembler/MVS & VM & VSE: Language Reference</i>	SC26-4940
<i>IBM High Level Assembler/MVS & VM & VSE: Programmer's Guide</i>	SC26-4941
<i>OS/VS2 MVS System Programming Library: Job Management</i>	GC28-1303
<i>Systems Application Architecture, An Overview</i>	GC26-4341
<i>Using Magnetic Tape Labels and File Structure</i>	SC26-4565
<i>VM/SP CMS Command Reference</i>	SC19-6209
<i>VM/SP System Messages and Codes</i>	SC19-6204
<i>VM/ESA: ESA/Extended Configuration Principles of Operation</i>	SC24-5594
<i>VM/XA SF CP Command Reference</i>	GC19-6215

Industry Standards

The VS FORTRAN Version 2 compiler and library are designed according to the specifications of the following industry standards, as understood and interpreted by IBM as of December 1990.

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

CLOCK supports the Industrial Real Time Fortran (IRTF) standards.

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. In the hardcopy manual, these extensions are printed in color, as in the following sentence:

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

Parallel Language

VS FORTRAN's implementation of parallel language tracks a subset of the PCF proposal (a precursor to ANSI committee X3H5 - Parallel Processing Constructs for High Level Programming Languages) as it advances through industry acceptance and the standards proposal process. Any future changes made by these groups to the parallel language that is implemented in VS FORTRAN may result in corresponding changes in future releases of the VS FORTRAN product, even if these changes result in incompatibilities with previous releases of the product.

Operating System Support

In this manual, MVS refers to MVS/SP*, MVS/XA, and MVS/ESA systems; VM refers to VM/SP, VM/XA*, and VM/ESA* systems; and XA refers to VM/XA, VM/ESA, MVS/XA, and MVS/ESA systems.

Support for extended common blocks requires execution of the compiled code under MVS/ESA or VM/ESA.

The following are supported under MVS only:

- Multitasking facility
- Data-in-virtual
- Asynchronous input/output

* MVS/SP, VM/ESA, and VM/XA are trademarks of the International Business Machines Corporation.

Summary of Changes

Release 6, October 1993

Major Changes to the Product

Support for AIX^{*}/370 is not included in VS FORTRAN Version 2 Release 6.

Support for AIX/ESA^{*} is found in the AIX VS FORTRAN/ESA product.

- Printable copies of certain of the VS FORTRAN Version 2 Release 6 publications are provided on the product tape.
- Support for additional language has been added, much of which address previous incompatibilities between XL FORTRAN and VS FORTRAN
 - Additional constant and operator support:
 - Quote-delimited literal character constants
 - Maximum negative integer literal value
 - Typeless constants (binary, octal and hexadecimal)
 - Named constants in complex constants
 - .XOR. logical operator
 - <> graphical relational operator
 - Storage classes:
 - STATIC and AUTOMATIC storage classification statements
 - IMPLICIT STATIC and IMPLICIT AUTOMATIC statement support
 - Additional data type support:
 - LOGICAL*2 and LOGICAL*8
 - INTEGER*1 and INTEGER*8
 - UNSIGNED*1
 - BYTE
 - DOUBLE COMPLEX and DOUBLE COMPLEX FUNCTION type specifications
 - XREF and MAP processing has been improved to remove storage-ordering perturbations.
 - Additional intrinsic function support:
 - INTEGER*8, INTEGER*2, and INTEGER*1 support for existing functions
 - XOR, LSHIFT, RSHIFT, ISHFTC, IBITS, and LOC have been added.
 - HALT compiler option to reduce compilation times for unsuccessful compilations
 - MVBITS and ARGSTR service routines
 - Dynamic storage support, for allocating and deallocating storage at run-time
 - Integer POINTER data type

^{*} AIX and AIX/ESA are trademarks of the International Business Machines Corporation.

- Including dynamically dimensioned arrays
- ALLOCATED, DEALLOCATE, and NULLIFY statements
- Intrinsic function ALLOCATED
- Compile-time option DDIM
- Dynamically loaded module support (via DYNAMIC option)
- I/O features enhancements:
 - Support for dummy arguments in NAMELIST lists.
 - NML keyword for NAMELIST READ/WRITE statements
 - OPEN and INQUIRE statement support for the POSITION, PAD, and DELIM specifiers
 - B and O format control codes
 - Expansion of the Z format control code
 - \$ format control code
- Optimization improvements:
 - Optimization for pipelined instruction scheduling
 - Improved optimization for Inter-loop register assignments
- Message improvements:
 - All messages written to SYSTERM file
 - Information messages no longer marked as errors
- Run-time options improvements:
 - Abbreviations are now allowed.
 - Default I/O units are now user-specifiable with ERRUNIT, RDRUNIT, PRTUNIT, and PUNUNIT
- Vector support enhancements:
 - The VECTOR option defaults can be set at installation.
 - VECTOR suboptions MODEL and SPRECOPT.
 - Vectorized SIGN, ANINT, DNINT, NINT and IDNINT intrinsic functions
 - Vector performance improvements
- Parallel support enhancements:
 - LOCAL statement allowing arrays for Parallel Do / Parallel Sections
 - NAMELIST processing for parallel environment allows local variables
 - Support for user-generated parallel trace (including service routines PTWRIT and PTPARM and suboption TRACE)
 - Parallel execution controls (affinity control)
 - Load module support for routines invoked via SCHEDULE and PARALLEL CALL

Major Changes to the Product

- Support has been added in the compiler and library for the Advanced Interactive Executive/370 (AIX/370) operating system. In addition, programs designed to be run on AIX/370 may use the following enhancements to VS FORTRAN Version 2:
 - The **fvs** command to invoke the VS FORTRAN Version 2 compiler.
 - Use of tab characters in source programs.
 - Communication between Fortran programs and C programs.
 - Coding of indirectly recursive Fortran routines.
- Support for parallel programs¹ has been added for programs running under CMS and MVS. These enhancements support:
 - Automatically generating parallel code for DO loops using a compile-time option.
 - Explicit coding of parallel loops, sections, and calls with parallel language extensions.
 - Using lock and event services to control synchronization in parallel programs.
 - Directing the compiler to generate parallel or serial code using enhanced directives.
 - Determining the number of virtual processors available and specification of the number of virtual processors to use during run time.
 - Using I/O within parallel programs.
 - Calling subroutines within parallel loops and sections.
 - Obtaining information for tuning your parallel program using a compiler report listing.
- Support for extended common blocks has been added for programs running under MVS/ESA or VM/ESA. Compile-time and run-time enhancements and options have been added to support the three types of common blocks that now exist.
- Virtual storage constraint relief has been added for compiling under MVS/XA, MVS/ESA, VM/XA, or VM/ESA. This support allows the compiler to reside above the 16MB line and to process in 31-bit addressing mode. Therefore, larger programs can now be compiled.
- The default name for a main program has been changed from MAIN to MAIN#, which allows MAIN to be used as a user name for a common block or other global entity.

¹ The VS FORTRAN Version 1 standard math routines (VSF2MATH) are not supported for parallel processing. Interactive debug can be used only with non-parallel programs, and with serial portions of parallel programs when the run-time option PARALLEL(1) is specified.

- Array declarator expressions for object-time dimensions can now be of type Integer*2.

Release 4, August 1989

Major Changes to the Product

- Enhancements to the vector and optimization features of VS FORTRAN Version 2
 - Automatic vectorization of user programs is enhanced by improvements to the dependence analysis done by the compiler. Specifically, the following constructs are eligible for vectorization:
 - Loops containing simple READ, WRITE, and PRINT statements
 - Loops containing equivalenced arrays
 - Loops containing branches out of the loop
 - Loop bound appearing in a subscript expression
 - Loop nests that process a triangular matrix
 - Simple IF loops
 - Integer sum reduction.
 - Additional advanced vector optimization.
 - In programs optimized at either OPT(2) or OPT(3), arithmetic constants will be propagated globally for scalar variables.
 - In programs optimized at either OPT(2) or OPT(3), informational messages are issued when local scalar variables may be referenced before they have been initialized.
 - Improved performance when the CHAR, ICHAR, and LEN character intrinsic functions are used.
 - Single and double precision complex divide routines can be vectorized.
- Enhancements to input/output support in VS FORTRAN Version 2
 - Improved data transfer rate for sequential DASD and tape input/output on MVS systems.
 - Ability to perform data striping (parallel I/O) on sequential data sets.
 - Ability to detect input conversion and record length errors.
- Enhancements to the intercompilation analyzer (ICA)
- Enhancements to the language capabilities of VS FORTRAN Version 2
 - Ability to use graphic relational operators as an alternative to FORTRAN 77 relational operators.
- Enhancements to the pseudo-assembler listing provided by VS FORTRAN Version 2
- Enhancements to the math routines:
 - Single, double, and extended precision MOD library routines are more precise.
 - Single and double precision scalar complex divide routines are more precise and faster, and are vectorizable.

- The old complex divide and MOD routines are in the alternate math library.

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, adjustably dimensioned arrays, and 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 online HELP function of interactive debug.
 - Vectorization is allowed at OPTIMIZE(2) and OPTIMIZE(3).
- 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 ENDDO 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 and I/O, and for interactive debug input and output.

* Systems Application Architecture is a trademark of the International Business Machines Corporation.

- 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 and MVS/ESA.
 - 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

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 an intercompilation analyzer. The ICA compile-time option invokes this analysis during compilation.
- 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.
- The ability, using the INQUIRE statement, to report file existence information based on the presence of the file on the storage medium.
- Addition of the OCSTATUS run-time option to control checking of file existence during the processing of OPEN statements, and to control whether files are deleted from their storage media.
- Under MVS, addition of a data set and an optional DD statement to be used during processing for loading library modules and interactive debug.
- Under VM, the option of creating during installation a single VSF2LINK TXTLIB for use in link mode in place of VSF2LINK and VSF2FORT.
- The ability to sample CPU use within a program unit using interactive debug. The new commands LISTSAMP and ANNOTATE have been added to support this function.

- The ability to automatically allocate data sets for viewing in the interactive debug source window.

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 run-time option
- Addition of support for VM/XA System Facility Release 2.0 (5664-169) operating system

Part 1. Language Reference

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 14. |
| Expressions | Executable sets of arithmetic, character, logical, or relational data. See Chapter 3, "Expressions" on page 32. |
| Statements | Combinations of data and expressions. See Chapter 4, "Statements" on page 47. |

Valid and Invalid Programs

This manual defines the syntax, semantics, and restrictions applicable for writing valid Fortran programs. Most violations of language rules are diagnosed by the compiler; however, some syntactic and semantic combinations cannot be diagnosed until run time.

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 can contain any statement except BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY, or RETURN.

Subprogram

Functions, subroutines, and block data program units are referred to generically as subprograms.

Function

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 can contain any statement except PROGRAM, SUBROUTINE, and BLOCK DATA.

Subroutine

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

Block data

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

Procedure

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

Intrinsic function

A function that performs mathematical, character, logical, or bit-manipulation operations. (See “INTRINSIC Statement” on page 163 and Chapter 6, “Intrinsic Functions” on page 280).

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.

Action statement

An executable statement that is not part of a construct. The executable statements DO, DO WHILE, and END DO; block IF, ELSE IF, ELSE, and END IF; PARALLEL DO, DOBEFORE, DOAFTER, DOEVERY, LOCAL, EXIT, and END DO; PARALLEL SECTIONS and END SECTIONS are all parts of larger syntactic constructs (do loops, for example). The other executable statements—assignment statements, input/output statements (except for FORMAT), allocation statements, the remaining control statements (unconditional GO TO, computed GO TO, assigned GO TO, arithmetic IF, logical IF, CONTINUE, STOP, PAUSE, END, CALL, RETURN), the remaining parallel statements (ORIGINATE, TERMINATE, PARALLEL CALL, WAIT FOR ALL CALLS, WAIT FOR ALL TASKS, WAIT FOR ANY TASK, SCHEDULE), and certain debug statements (DISPLAY, TRACE OFF, and TRACE ON) —are called *action statements*.

Preconnected file or unit

A file or unit is preconnected if it is connected before running a program that references the unit. This allows the program to perform input/output operations without first using an OPEN statement. For further information on connection and preconnection,

see “Input/Output Semantics” on page 49. The terms file connection and unit connection are equivalent.

Additional definitions can be found in the *VS FORTRAN Version 2 Master Index and Glossary*.

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 source input that can be 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 compile-time 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 character set, shown in Figure 3 on page 5, 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
\$ Currency symbol					= Equals sign
					" Quotation mark
					< Less than
					> Greater than

Figure 3. Source Statement Characters (Character Set)

The EBCDIC and ASCII code point tables, Figure 4 on page 6 and Figure 5 on page 7, show the two-digit hexadecimal code points required by the Fortran compiler for the characters in the character set. To determine the hexadecimal code point of a character in either table, look at the top of the column for the first digit, and at the left of the row for the second digit.

The character set also includes the shift-out character <shift-out> and the shift-in character <shift-in>. The empty spaces in Figure 4 on page 6 and Figure 5 on page 7 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 in 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 maximum number of characters allowed in a statement

As many blanks as desired can be included in a statement or comment to improve readability. They are ignored by the compiler.

Figure 4. EBCDIC Source Statement Code Points												
Second Hex Digit	First Hexadecimal Digit											
	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0			-									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				"								

Double-byte characters can be used in comment lines, names, and character constants when the DBCS compile-time option is in effect.

The term character, when used without a modifier, refers to a one-byte object. A double-byte character occupies two character storage units.

Note for DBCS Representation in Source: The DBCS compile-time option must be in effect for double-byte character text to be interpreted correctly. For information on the DBCS compile-time option, see *VS FORTRAN Version 2 Programming Guide*.

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').

Figure 5. ASCII Source Statement Code Points

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

Names

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

- An array (see “Arrays” on page 26)
- A variable (see “Variables” on page 23)
- A constant (see “PARAMETER Statement” on page 176)
- A main program (see “PROGRAM Statement” on page 182)
- A statement function (see “Statement Function Statement” on page 225)
- An intrinsic function (see “INTRINSIC Statement” on page 163)
- A function subprogram (see “FUNCTION Statement” on page 131)
- A subroutine subprogram (see “SUBROUTINE Statement” on page 229)
- A block data subprogram (see “BLOCK DATA Statement” on page 65)
- A common block (see “COMMON Statement” on page 73)
- An external user-supplied subprogram that cannot be classified by its usage in a program unit as either a subroutine or function subprogram name (see “EXTERNAL Statement” on page 103)
- A NAMELIST (see “NAMELIST Statement” on page 164)

Global Names

Global names must consist of EBCDIC characters.

Classes of global names are:

- Common block
- External function
- Subroutine
- Main program
- Block data

If a global name is longer than seven characters, the first four and last three characters are used to form the external symbol. The external symbol is then used to identify the global entity. For example, these names:

```
PASSED_PARAMETER  
HASH$FUNCTION
```

would be shortened to the following:

```
PASSTER  
HASHION
```

In MVS the external symbol must not contain an underscore character.

Local Names

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

Classes of local names are:

- Array
- Variable
- Constant
- Statement function
- Intrinsic function
- Dummy procedure
- NAMelist

Note: Local names are not equivalent to variables specified to be local by the LOCAL statement. The LOCAL statement is used during parallel processing. See “LOCAL Statement (for Parallel Loops)” on page 270 and “LOCAL Statement (for Parallel Sections)” on page 275.

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 name, a subroutine name, a block data program-unit 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.

EBCDIC and DBCS Names

Definitions

EBCDIC Name— In FORTRAN 66, sequence of 1 to 6, or in FORTRAN 77 extension, 1 to 31 letters, digits, or underscore characters contained in the EBCDIC character set. The first character 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 Names:

X_ray (An EBCDIC name)

<shift-out>kk.1.2.3 <shift-in> (A DBCS name)

Source Language Statements

The rules for forming each type of source language statement are defined in Chapter 4, "Statements" on page 47, which also contains a description of each 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 can 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. However, it is not recommended for use. 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 inline comment begins with an exclamation point, which initiates a comment anywhere on a line except when the exclamation point appears within character context. This inline comment can be used on free-form source lines.

The following figure illustrates the use of comment lines in free-form input.

```
Column: 1      7
        "THIS COMMENT BEGINS THE PROGRAM
        .
        .
10D=010.5
GOTO 56      ! UNCONDITIONAL BRANCH
150 A=B+C*(D+E**F-- ! THE FIRST '-' AFTER F
-! IS THE SIGN FOR SUBTRACTION, THE SECOND '-' INDICATES THE-
-! LINE IS TO BE CONTINUED.-
G+H-2.*(G+P))
! THE '-' AT THE BEGINNING OF THE NEXT LINE INDICATES IT-
-! IS A CONTINUED LINE
! THIS IS ANOTHER COMMENT
        .
        .
END
```

Figure 6. Valid Comment Placement

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. Columns 73 through 80 are considered part of the statement text in free form and cannot be used for identification.

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.

Initial Line

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

Continued Lines

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

When an inline 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 6 for an example.

Preserving a Minus Sign

A minus sign as the last character in a line indicates continuation. If the last two characters in a line are minus signs, the last one is a continuation character, and the preceding one is preserved as a minus sign. See Figure 6 for an example.

Continuation Lines

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

In Figure 7, HEAD_LINE1 and HEAD_LINE2 are equivalent:

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

Figure 7. Shift-out/Shift-in Characters in Continued Source

Figure 8 illustrates free-form source statements.

```
Column:  1
-----
@PROCESS FREE
"SAMPLE TEXT
.
.
CHARACTER*200  BLINE0
IF (IREAD.GE.3) THEN                                ! Get ready for next line
IF (BLINE0(1:1).NE.'$') THEN
IWRITE = IWRITE - 1
WRITE (UNIT=7,FMT=200) BLINE0, BLINE1, BLINE2, BLINE3, -
                        TOKEN1, TOKEN2, TOKEN3, TOKEN4, -
                        PGMNAME, LOPNAME
ENDIF
WRITE (UNIT=6,FMT=300) BLINE0                        ! Don't process tokens
ENDIF
.
.
```

Figure 8. 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 within a character or Hollerith constant or in column 6 (where it is treated as a continuation character). This inline comment consists of the exclamation point and all the characters to its right, up to the end of the line. The inline comment is treated as a blank character.

Comment lines can appear anywhere in a program unit before the END statement, including before a continuation line. Blank lines can appear anywhere in a program unit and are processed as comment lines. Comments can 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 can continue on as many as 99 continuation lines. Multiple statements per line are not allowed. Every statement in a program unit can 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 can be used for identification, sequencing, or any other purpose. A statement is terminated by another statement or by the end of the input.

Initial Line

Column 6 of the initial line of a statement must be a blank or a zero. The initial line of every statement can 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 can 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 can contain characters, but they are ignored.

In Figure 9, HEAD_LINE1 and HEAD_LINE2 are equivalent:

Column:	1	6		72

		CHARACTER*50 HEAD_LINE1		
		CHARACTER*50 HEAD_LINE2		
		HEAD_LINE1 = '<shift-out>A .H.E.A.D.I.N.G<shift-in>		
		X<shift-out> .L.I.N.E<shift-in>'		
		HEAD_LINE2 = '<shift-out>A .H.E.A.D.I.N.G .L.I.N.E<shift-in>'		

Figure 9. Shift-Out/Shift-In Characters in Continued Source Lines

Figure 10 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
		15	A = B + C * (D + E ** F +			SAMP0230
		1	G + H - 2. * (G + P))			SAMP0240
			C = 3.			SAMP0250
			.			.
			.			.

Figure 10. Example of Fixed-Form Source Statements

Statement Labels

Statement labels uniquely identify statements within a program unit and can be written in either free-form or fixed-form. The initial line of a statement can contain a label as the first (leftmost) entry on the line. A label can 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.

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

For detailed information on how to use statement labels in the two input formats see “Free-Form Input Format” on page 9 and “Fixed-Form Input Format” on page 11.

Keywords

Keywords identify VS FORTRAN Version 2 intrinsic functions and utility procedures, which can be used as part of any program. 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. For more information, see Chapter 6, “Intrinsic Functions” on page 280, and Chapter 8, “Service Subroutines” on page 318.

Chapter 2. Data

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 five types of variables, as discussed under “Data Types and Lengths” on page 23.

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.
- *Binary, Octal, and Hexadecimal* constants, known collectively as *boz* constants, can be used for data initialization and in expressions. Boz constants are also referred to as *typeless* constants. See “Using Context-Specific Data Types in Expressions” on page 46 for information on using these constants.

You can name a constant with the PARAMETER statement. (See “PARAMETER Statement” on page 176.)

Arithmetic Constants

Arithmetic constants fall into three categories: integer, real, and complex. Arithmetic constants can be unsigned or signed. 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 Positive Value:

Integer*1 127 (that is, $2^{**7} - 1$)

Integer*2 32767 (that is, $2^{**15} - 1$)

Integer*4 2147483647 (that is, $2^{**31} - 1$)

Integer*8 9223372036854775807 (that is, $2^{**63} - 1$)

Maximum Negative Value:

Integer*1 -128 (that is, -2^{**7})

Integer*2 -32768 (that is, -2^{**15})

Integer*4 -2147483648 (that is, -2^{**31})

Integer*8 -9223372036854775808 (that is, -2^{**63})

An integer constant is 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 absolute value 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

-2147483648

Invalid Integer Constants:

27. Contains a decimal point.

65E6 Contains an exponent

5,396 Contains an embedded comma.

Real Constants

Definition

Real Constant—A string of decimal digits that expresses a real number. It has 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.

Absolute value:

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 is 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 can 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	Absolute value outside the allowable range, that is, 23.5 x 10**97 > 16**63	..
21.3D-99	Absolute value 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 is positive, zero, or negative and must be within the allowable range. If unsigned and nonzero, they are assumed to be positive. A zero is 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.

Named constants can be used in complex constants.

Valid Complex Constants: (i = <sqrt of -1>)

(3,-1.86)	Has the value 3.0 - 1.86i; both parts are real (4 bytes long).
-----------	--

REAL*4 APPLE,ORANGE	Has the value 3.0 - 1.86i;
PARAMETER(APPLE = 3.0)	both parts are real (4 bytes
PARAMETER(ORANGE = -1.86)	long) and expressed as named
(APPLE,ORANGE)	constants.

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

Invalid Complex Constants:

<code>DATA A/1.4/ (A, 3.7)</code>	Real part is not a constant.
<code>(.0009Q-1, 7643.Q+1199)</code>	Too many digits in the exponent of the imaginary part.
<code>(49.76, .015D+92)</code>	Absolute value 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 41.)

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” on page 119 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 or quotation marks.

Character constants can consist 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 can 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 , PARALLEL CALL or SCHEDULE statement or function reference
- An input or output statement
- A FORMAT statement
- A PARAMETER statement
- A PAUSE or STOP statement

The delimiting apostrophes or quotation marks are not part of the data represented by the constant. An apostrophe within apostrophe-delimited character data is represented by two consecutive apostrophes, with no intervening blanks. Similarly, a quotation mark within quotation-mark-delimited character data is represented by two consecutive quotation marks, with no intervening blanks. In a character constant, blanks embedded between the delimiters (apostrophes or quotation marks) are significant.

The length of a character constant is the number of character storage units needed to represent the character data between the delimiters. (Consecutive apostrophes or quotation marks representing one character 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
"DON'T"	5
'<shift-out>.D.O.N.' 'T<shift-in>'	12
'<shift-out>.D.O.N.' 'T<shift-in>DO'	14
'<shift-out><blank><blank><shift-in>'	4

Hollerith Constants

Definition

Hollerith Constant—A string of any characters capable of representation in the processor and preceded by wH, 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

Binary Constants

Definition

Binary Constant—A string of binary digits (zeroes and ones), enclosed in apostrophes or quotation marks and preceded or followed by the letter B.

A binary constant can be used as an argument, as a data initialization value for any type of variable or array, or in an arithmetic, logical, or character expression.

Binary constants can be of any length, with these two qualifications:

- If the number of characters in the binary constant is not a multiple of 8, the value is padded on the left with zeroes.
- If the value of a binary constant is larger in bytes than the data object in which the value is placed, the value is truncated on the left. The truncated bits are lost.

Valid Binary Constants

Binary Constant	Integer Value
B'1011'	11
B'0111'	7
"1111111"B	127

Octal Constants

Definition

Octal Constant—A string of octal digits formed from the set of characters 0 through 7, enclosed in apostrophes or quotation marks and preceded or followed by the letter O.

An octal constant may be used as an argument, as a data initialization value for any type of variable or array, or in an arithmetic, logical, or character expression.

Octal constants can be of any length.

- If the number of bits in the octal constant is not a multiple of 8, the value is padded on the left with zero bits.
- If the value of an octal constant is larger in bytes than the data object in which the value is placed, the value is truncated on the left. The truncated bits are lost.

Valid Octal Constants

Octal Constant	Bit String	Integer Value
0'123'	01010011	83
0'1000'	00000010 00000000	512
"77"0	00111111	63

Hexadecimal Constants

Definition

Hexadecimal Constant—A string of hexadecimal digits formed from the set of characters 0 through 9 and A through F or a through f, enclosed in apostrophes or quotation marks and preceded or followed by either the character X or the character Z.

As a data initialization value only, a hexadecimal constant beginning with Z can omit apostrophes or quotation marks.

A hexadecimal constant can be used as an argument, as a data initialization value for any type of variable or array, or in an arithmetic, logical, or character expression.

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

Hexadecimal constants can be of any length.

- If the number of characters in the hexadecimal constant is not a multiple of 2, the value is padded on the left with zeroes.
- If the value of a hexadecimal constant is larger in bytes than the data object in which the value is placed, the value is truncated on the left. The truncated bits are lost.

Valid Hexadecimal Constants

Hexadecimal Constant	Bit String	Integer Value	Comments
Z1C49A2F1 Z'1C49A2F1' "1C49A2F1"X	00011100 01001001 10100010 11110001	474,587,889	The first form is valid only in DATA statements.
X"BADFADE" 'BADFADE'Z	00001011 10101101 11111010 11011110	195,951,326	The first 4 zero bits are implied because an odd number of hexadecimal digits are written.

Invalid Hexadecimal Constants

XABC	Needs quotes or apostrophes, or change X to Z.
101Z	Needs quotes or apostrophes, or Z must come first.

Using Typeless Constants

The data type and length of a typeless constant are determined by the context in which you use the typeless constant. VS FORTRAN does not convert them before use.

- When you use such a constant with an arithmetic or logical unary operator, the constant assumes an INTEGER*4 data type.
- When you use such a constant with an arithmetic, logical, or relational binary operator, the constant assumes the same data type as the other operand. If both operands are typeless constants, they assume an INTEGER*4 data type.
- When you use such a constant in a concatenation operation, the constant assumes a CHARACTER data type.
- When you use such a constant as the expression on the right-hand side of an assignment statement, the constant assumes the type of the variable, array element, or character substring on the left-hand side.
- When you use such a constant in a context that requires a specific data type, the constant assumes that data type.
- When you use a typeless constant as an initial value in a DATA, STATIC, or explicit type statement, or as the constant value of a named constant in a PARAMETER statement, or when the typeless constant is to be treated as any noncharacter type of data, the following rules apply:
 - If a boz constant is smaller than the length expected, the VS FORTRAN compiler adds zeros on the left; if it is longer, the compiler truncates on the left.
 - If a typeless constant specifies the value of a named constant with a CHARACTER data type having inherited length, the named constant will have length equal to the number of bytes specified by the named constant.
- When a typeless constant is to be treated as an object of type CHARACTER (except when used as an initial value or as the constant value of a named constant), the length is determined by the number of bytes specified by the typeless constant.
- When you use such a constant as part of a complex constant, the constant assumes the data type of the other part of the complex constant. If both parts

are typeless constants, the constants assume the real data type with length sufficient to represent both typeless constants.

- When you use such a constant as an actual argument, the type of the corresponding dummy argument must be an intrinsic data type; the dummy argument must not be a procedure, array, or alternate return specifier.

Variables

A variable is a data item, identified by a name, that occupies a storage area. (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 is 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. However, if a variable is specified on a LOCAL statement after having been assigned a value, the local instances of that variable are not initialized with that value.

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

<shift-out> kk.1.2.3 <shift-in>

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 11 on page 24.) 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 boz 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 55.)

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 can use an optional length specification.

Figure 11 shows each data type with its associated storage length and standard length.

Figure 11. Data Types and Valid Lengths

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

A programmer declares 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 is declared only by explicit or IMPLICIT specification statements. If no length specification is made in these statements, the default length is assumed. INTEGER, BYTE, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, LOGICAL, and CHARACTER are used to specify the length and type in these statements.

VS FORTRAN Version 2 accepts:

- BYTE to indicate 1 byte with a value in the range of -128 to 127
- INTEGER*1 to indicate 1 byte, with a value in the range of -128 to 127
- INTEGER*2 to indicate 2 bytes
- INTEGER*4 as an alternative to INTEGER, to indicate 4 bytes
- INTEGER*8 to indicate 8 bytes
- UNSIGNED*1 to indicate 1 byte, with a value in the range of 0 to 255
- 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*2 to indicate 2 byte
- LOGICAL*4 to indicate 4 bytes, as an alternative to LOGICAL
- LOGICAL*8 to indicate 8 byte

- 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, as an alternative to DOUBLE COMPLEX (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 first 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 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. You can use the IMPLICIT statement 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 143.

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 99.

Arrays

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 and can have 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, you must specify the array name and a subscript. 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 32.)

- Subscript expressions may contain arithmetic expressions that use any of the arithmetic operators: +, -, *, /, **.
- Subscript expressions may contain function references that do not change any other value in the same statement.
- Subscript expressions may contain array elements.
- 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.
- 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 27.)

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 of length 2 or 4.

- 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.
- The constant expression can contain the intrinsic functions ICHAR and LEN. ICHAR and LEN must have constant arguments. The argument to LEN must not be a dummy argument that has been specified with an inherited length. These are the only function references that are allowed in dimension bound expressions.
- 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 three kinds of arrays:

- An *actual* array is one whose name is not a dummy argument and whose dimension bound expressions is a constant expression of type integer.
- A *dummy* array is one whose name must contain a dummy argument and whose dimension bound expressions can also contain 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.

- A *pointee array* is one that is associated with and addressed by a pointer and whose dimension bound expressions can also contain integer variables that are not necessarily dummy arguments or entities in a common block. It is important to understand that, in a real sense, pointee arrays do not exist, but rather serve as templates for mapping storage obtained by allocation or by assigning the associated pointer to the address of existing storage.

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 arrays in a program, so that an appropriate amount of storage can be reserved (for actual arrays), allocated (for pointee arrays), and addressed (particularly for dummy and dynamically dimensioned arrays). Only one declaration of an array is permitted in a program unit.

Declaration of this information is made by one of the following:

- A DIMENSION statement (see “DIMENSION Statement” on page 85)
- A COMMON statement (actual arrays only - see “COMMON Statement” on page 73)
- An explicit type statement (see “Explicit Type Statement” on page 99)
- A POINTER statement (pointee arrays only - see “POINTER Statement” on page 178)
- A storage-class statement (actual arrays only - see “AUTOMATIC Statement” on page 63 and “STATIC Statement” on page 228)

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. Each character array element is treated as a single entity. Portions of a character array element can be accessed through substring notation.

Object-Time Dimensions

The absolute dimensions of a dummy 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 can contain dummy arguments or variables in a common block that are integer variables of length 4 or 2, to specify the size of the array. This is known as an “adjustable dimension declarator”

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.

Dynamically Dimensioned Arrays

When the dimensions of a pointee array are not constant expressions, the array is *dynamically dimensioned*. The array declarators can contain variables of type integer (length 2 or 4) whose values are known at the time the array is referenced.

When the declarator expressions are evaluated depends on the DDIM compile-time option. When the NODDIM compiler option is specified, pointee arrays have their dynamic dimensions evaluated upon entry to the containing subprogram; these dimensions are then fixed for the duration of that execution of the program unit.

When the DDIM compiler option is specified, all pointee arrays in the program unit have their dynamic dimensions evaluated at each array element reference. This

allows the values specified for the dimensions to be changed during program execution; each reference to any array element after the dimension has been changed will reference the array using the new dimension specifications.

Notice that dynamically dimensioned arrays may be used in the main program only if compiled with the DDIM option and that, in subroutines and functions, the dimensions must be dummy arguments or in Common if compiled with the NODDIM option.

For additional considerations when used in a parallel loop or parallel section, see “LOCAL Statement (for Parallel Loops)” on page 270 and “LOCAL Statement (for Parallel Sections)” on page 275, respectively.

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 “Arrays” on page 26).

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 340.

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 = 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 = '<shift-out>I.B.M.<shift-in>'
FRONT = LOGO(1:4)
```

then FRONT has the value <shift-out>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 340.

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, complex, or unsigned.
- 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:

- For a result to be defined, 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 55.) 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 can 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 expression is the type of the primary. Examples of arithmetic expressions are shown in Figure 12.

Figure 12. Examples of Arithmetic Expressions

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

Arithmetic Operators

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

The arithmetic operators are shown in Figure 13.

Figure 13. Arithmetic Operators

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

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 can 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 14.

Figure 14. Hierarchy of Arithmetic Operations

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

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 15 on page 36 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 or unsigned 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) cannot have a real exponent.

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

When one operand is unsigned and the other is integer or byte, the unsigned operand is converted to the other operand, padded, if necessary, on the left with zeroes, and given a positive sign.

When one operand is unsigned and the other is real or complex, the unsigned operand is converted to a positive INTEGER*4 value, which is then converted to the type and length of the other operand.

Figure 15 shows the type and length of the result of adding, subtracting, multiplying, or dividing two arithmetic operands.

Figure 15 (Page 1 of 2). Type and Length of Result with Mixed-type Operands

One Operand	Other Operand	Result
Byte	Byte	Byte
	Unsigned*1	Integer*1
	Integer*1	Integer*1
	Integer*2	Integer*2
	Integer*4	Integer*4
	Integer*8	Integer*8
	Real*4	Real*4
	Real*8	Real*8
	Real*16	Real*16
	Complex*8	Complex*8
	Complex*16	Complex*16
	Complex*32	Complex*32
Unsigned*1	Unsigned*1	Unsigned*1
	Integer*1	Integer*1
	Integer*2	Integer*2
	Integer*4	Integer*4
	Integer*8	Integer*8
	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*1	Integer*1	Integer*1
	Integer*2	Integer*2
	Integer*4	Integer*4
	Integer*8	Integer*8
	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*2	Integer*2	Integer*2
	Integer*4	Integer*4
	Integer*8	Integer*8
	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*4	Integer*4
	Integer*8	Integer*8
	Real*4	Real*4
	Real*8	Real*8
	Real*16	Real*16
	Complex*8	Complex*8
	Complex*16	Complex*16
Integer*8	Integer*8	Integer*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

Figure 15 (Page 2 of 2). Type and Length of Result with Mixed-type Operands

One Operand	Other Operand	Result
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	Real*8	Real*8
	Real*16	Real*16
	Complex*8	Complex*16
	Complex*16	Complex*16
	Complex*32	Complex*32
Real*16	Real*16	Real*16
	Complex*8	Complex*32
	Complex*16	Complex*32
	Complex*32	Complex*32
Complex*8	Complex*8	Complex*8
	Complex*16	Complex*16
	Complex*32	Complex*32
Complex*16	Complex*16	Complex*16
	Complex*32	Complex*32
Complex*32	Complex*32	Complex*32

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 can 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.

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.

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 can 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 can be formed by using one or more character operands, together with character operators and parentheses.

Concatenation Operation in Character Expressions

The character operator is shown below.

Character Operator	Definition
//	Concatenation

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 99.

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'.

Constant Expressions

A constant expression is an arithmetic, character, logical, or relational expression in which all the operands are constants, either named or unnamed. The results of the intrinsic functions CHAR, ICHAR, and LEN are considered to be constants, if their arguments are constants.

Relational Expressions

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

The relational operators are shown in Figure 16.

Figure 16. Relational Operators

Standard	VS FORTRAN	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
	<>	

Relational operators:

- Express a condition that can be either true or false.
- Can be used to compare two arithmetic expressions (except complex) or two character expressions. Only the .EQ. (==) and .NE. (/= or <>) operators can 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 15 on page 36, except when one operand is unsigned and the other is real or complex. In that case, the unsigned operand is converted first to INTEGER*4.
- 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 4 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.

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 <= (5*ROOT+4)

.5 .GE. (.9*ROOT)

E == 27.3E+05

CHAR .EQ. 'ABCDEFGH'

C /= 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==(A+F)	Logical quantities can never be compared by relational operators.
E**2 .LT 97.1E1	There is a missing period immediately after the relational operator.
>9	There is a missing arithmetic expression before the relational operator.
E*2 .EQ. 'ABC'	A character expression cannot 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*2, or LOGICAL*8.

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 can 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 can be formed by using logical operators to combine logical primaries.

Logical Operators

The logical operators are shown in Figure 17 on page 42. (A and B represent logical constants or variables, or expressions containing relational operators.)

Figure 17. Logical 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. .XOR.	A.NEQV.B A.XOR.B	If A and B are both true or both false, then A.NEQV.B (or A.XOR.B) is false; otherwise, it is true.

The only valid sequences of two logical operators are:

.AND..NOT.

.OR..NOT.

.EQV..NOT.

.NEQV..NOT.

.XOR..NOT.

The sequence .NOT..NOT. is invalid.

Only those expressions that have a value of true or false when evaluated, can 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 18. Within a hierarchic level, computation is performed from left to right.

Figure 18. Hierarchy of Operations Involving Arithmetic Operators

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. or .XOR.	9th

In the evaluation of logical expressions, priority of operations involving *character* operators is as shown in Figure 19 on page 44. Within a hierarchic level, computation is performed from left to right.

Figure 19. Hierarchy of Operations Involving Character Operators

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. or .XOR.	7th

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 can 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 following:

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.

Using Context-Specific Data Types in Expressions

Hexadecimal, octal, and binary constants are typeless constants.

Typeless (boz) constants and variables that are of type `BYTE` can be used wherever a `LOGICAL*1`, `CHARACTER*1`, or `INTEGER*1` named constant or variable can be used.

The data types of these two groups are determined by the context in which you use them. VS FORTRAN does not convert them before use.

Using Typeless Constants

Typeless constants are described in “Using Typeless Constants” on page 22.

Using BYTE Data Objects

- When you use a `BYTE` data object as an operand of an arithmetic, logical, or relational binary operator, the data object assumes:
 - An `INTEGER*1` data type if the other operand is arithmetic, `BYTE`, or a typeless constant.
 - A `LOGICAL*1` data type if the other operand is `LOGICAL`.
 - A `CHARACTER*1` data type if the other operand is `CHARACTER`.
- When you use a `BYTE` data object as an operand of the concatenation operator, the data object assumes a `CHARACTER*1` data type.
- When you use a `BYTE` data object as an actual argument, the data object assumes a length of 1 byte and no data type.
- When you use a `BYTE` data object in a context that requires a specific data type, which is arithmetic, `LOGICAL`, or `CHARACTER`, the data object assumes an `INTEGER*1`, `LOGICAL*1`, or `CHARACTER*1` data type, respectively.
- When you use a `BYTE` data object in any other context, the data object assumes an `INTEGER*1` data type.

Chapter 4. Statements

A source program is 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
- PARALLEL statements
- DATA statement
- Debug statements
- Input/output statements
- PROGRAM statement
- Specification statements
- Subprogram statements
- Compiler directives
- Allocation statements

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 within a parallel thread or non-parallel program are executed sequentially. Control statements can alter this normal sequence of execution of statements. Control statements are executable. The following are control statements:

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

Parallel Statements

The following are Parallel statements:

DOAFTER	ORIGINATE	TERMINATE
DOBEFORE	PARALLEL CALL	WAIT FOR ALL CALLS
DOEVERY	PARALLEL DO	WAIT FOR ALL TASKS

END SECTIONS	PARALLEL SECTIONS	WAIT FOR ANY TASK
EXIT LOCAL	SCHEDULE SECTION	WAIT FOR TASK

See Chapter 5, “Parallel Language Constructs” on page 264 for specific parallel statements and constructs.

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. It cannot be used with variables or arrays defined in extended common blocks, and is not recommended for use in programs. 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	DISPLAY	TRACE OFF
DEBUG	END DEBUG	TRACE ON

The static debug facility differs from interactive debug, described in *VS FORTRAN Version 2 Interactive Debug Guide and Reference*. Interactive debug 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	INQUIRE	REWIND
CLOSE	OPEN	REWRITE
DELETE	PRINT	WAIT
ENDFILE	READ	WRITE
FORMAT		

Note: The input and output statements are described 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

Input/output statements are based on a set of semantics that 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. All files in CMS and MVS 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.

VS FORTRAN uses reserved *ddnames* in CMS and MVS when referring to an unnamed file. Files referred to by these reserved names cannot be specified by name in a Fortran program. A named file is specified in a VS FORTRAN program by its *ddname* or by its CMS file identifier or MVS data set name.

Source files that specify system dependent file names in I/O statements can be compiled on any of the supported operating systems; however, the object file may not run. For example, if you compile a source file that contains a CMS file name in an I/O statement under CMS, and then transfer the object file to an MVS system, the object file will not run properly.

When running a parallel program, each parallel task has a unique I/O environment with its own set of units for I/O; that is, the units in one task are logically distinct from the units in another task. While the root task uses the same default *ddnames* for unnamed files as are used by a serial program (i.e.: FTnnFmmm, FTnnPmmm, etc.), originated tasks have their own set of default *ddnames* for the standard units. The default *ddnames* of the root task cannot be accessed from within an originated task, and vice versa. If a file is open in one task and an INQUIRE by filename is processed in another task, INQUIRE will report that the file does not exist for that task.

With dynamic file allocation on CMS or MVS, 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 VS 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 VS 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 are connected to units before running programs that reference the units. This allows programs to perform input/output operations on those files without first using an OPEN statement for each file. 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 can be preceded by a FORMAT or an ENTRY statement. Specification statements are nonexecutable.

The following are specification statements:

AUTOMATIC	EXTERNAL	POINTER
COMMON	IMPLICIT	PARAMETER
DIMENSION	INTRINSIC	SAVE
EQUIVALENCE	NAMelist	STATIC

Explicit type:

BYTE	DOUBLE COMPLEX	LOGICAL
CHARACTER	DOUBLE PRECISION	REAL
COMPLEX	INTEGER	UNSIGNED

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. Control is transferred to a subroutine when its name is referenced in a CALL statement. A PARALLEL CALL statement assigns a subroutine to run as a parallel thread. A SCHEDULE statement assigns a subroutine to run in a parallel task. The statements following the PARALLEL CALL or SCHEDULE can proceed without waiting for the subroutine to finish executing.

Block data subprograms begin with a BLOCK DATA statement. 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 and parallel directives, see *VS FORTRAN Version 2 Programming Guide*.

Allocation Statements

There are three allocation statements: ALLOCATE, DEALLOCATE, and NULLIFY. The ALLOCATE statement obtains space dynamically; the DEALLOCATE statement returns the space; and the NULLIFY statement disassociates a pointer from assigned space. Allocation statements are executable.

Order of Statements in a Program Unit

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

1. PROGRAM, or subprogram statement, if any.
2. IMPLICIT NONE statements, if any.
3. PARAMETER statements and/or IMPLICIT statements, if any.
4. Other specification statements, if any.
5. Data statements, see Figure 20 on page 52.
6. Statement function definitions, if any.
7. Executable statements and parallel statements.
8. END statement.

For the order of DEBUG statements, see “DEBUG Statement” on page 79.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements, except PARAMETER statements. Any specification statement that specifies the type of a 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 can appear anywhere after the PROGRAM or subprogram statement and before the END statement. The ENTRY statement, however, cannot 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 can be intermixed with the other specification statements. DATA statements for a variable or array must follow any specification statements that specify the data type or dimension information for that entity.

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

The order of statements in BLOCK DATA subprograms is discussed under “BLOCK DATA Statement” on page 65.

Figure 20 shows the order of statements.

- 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.

Figure 20. Order of Statements and Comment Lines

Comment Lines	PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement		
	FORMAT and ENTRY Statements	IMPLICIT NONE Statement	
		PARAMETER Statements	IMPLICIT Statements
			Other Specification Statements
		DATA Statements	Statement Function Statements
			Executable Statements and Parallel Statements
END Statement			

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.

@PROCESS Statement

The options you specify when you request compilation remain in force for all source programs you're compiling, unless you override them with the @PROCESS statement.

Each source program requires its own @PROCESS statement if you wish to override the options specified at compiler invocation. If any source program does not have its own @PROCESS statement, it is compiled according to the compiler-invocation specifications *not* according to the @PROCESS specifications preceding source program in the job stream.

If an option on the @PROCESS statement conflicts with an option from another source, for example, a default option or one you specify at the compiler invocation, see *VS FORTRAN Version 2 Programming Guide* for a listing of which option is assumed.

To code the compile-time options, place the @PROCESS statement as the first line in the source program. The following coding rules apply:

- @PROCESS must appear in columns 1 through 8 of the statement.
- The @PROCESS statement can be followed by compiler options in columns 9 through 72 of the statement. The options must be separated by commas or blanks.
- Up to 20 PROCESS statements can be supplied for a program unit. Columns 9 through 72 of a following @PROCESS statement are appended to the previous @PROCESS statement.

Intervening lines must not appear between @PROCESS statements. A line such as a blank, comment, or INCLUDE will terminate @PROCESS statement input, and subsequent @PROCESS statements will be ignored.

- A conditional INCLUDE option will not be overridden with another conditional INCLUDE specified on the @PROCESS statement. The conditional INCLUDE will be added to any others in effect.

The following restrictions apply to the options you may specify on @PROCESS:

- All compile-time options except OBJECT, DECK, [DISK, PRINT, and NOPRINT] are permissible. The latter three are available only to CMS users; for details, see *VS FORTRAN Version 2 Programming Guide*.
- If NODECK or NOOBJ has been specified on the compiler invocation, you cannot specify DECK or OBJ, respectively, on the @PROCESS statement.
- TERMINAL and TRMFLG cannot be specified on the @PROCESS statement if TERMINAL was not specified on the compiler invocation or in the installation defaults.
- Only the NOICA form of the ICA option can be specified in the @PROCESS STATEMENT.
- An @PROCESS statement must be placed before all other source statements in a compilation unit. Flexibility is provided by allowing an @PROCESS to be preceded by comment lines, EJECT statements, or certain INCLUDE directives.

However, if the following compile-time options appear in an @PROCESS statement that is preceded by comment lines, EJECT statements, or INCLUDE directives, then the option will be ignored and a level 4 message issued:

- DBCS
- SAA
- FIPS
- LANGLVL
- CHARLEN
- NAME
- VECTOR
- FREE|FIXED
- DIRECTIVE
- PARALLEL

ALLOCATE Statement

The ALLOCATE statement obtains space for a pointee array.

Syntax

ALLOCATE (*pointee1* [, *pointee2* ...], **STAT=***stat*)

pointee

is the name of the pointee array

stat

is the name of an INTEGER*4 variable that receives a status code. Possible status codes are:

- 0** The allocation request was successfully completed.
- 16** Insufficient storage was available to allocate all the pointee arrays; objects up to the one for which storage could not be obtained were allocated. (Use the ALLOCATED function to determine which arrays were allocated.)

If the ALLOCATE statement refers to a dynamically dimensioned array, compiler options determine the values of the dimension specifiers.

- If the DDIM compiler option is used, the value of the dimension specifiers at the time of allocation is used.
- If the NODDIM compiler option is used, the value of the dimension specifiers at the time of entry to the subprogram is used.

Storage assigned by an ALLOCATE statement remains assigned until it is explicitly freed by a DEALLOCATE statement, or until the main program ends.

If you allocate storage for a pointee array that already has storage allocated, the previously allocated storage remains allocated until the main program ends. Note that this storage is inaccessible unless it is also addressed by another pointer variable.

In a parallel environment, storage assigned by an ALLOCATE statement is used globally, not locally. If the pointer variable is local to the parallel execution unit, the storage will be inaccessible when the parallel execution unit terminates, unless another pointer variable outside the parallel execution unit addresses it.

Example 1:

```
Pointer (Cpt, C(10,10)), (Dpt, D(2,2))
Allocate (C, D, Stat=ir)
```

Example 2:

```
@PROCESS DDIM
Pointer (dynp, DArray(i,j))
Read(5,*) i, j
Allocate (DArray, Stat=ir)
```

Related Statements

- Use the DEALLOCATE statement (see “DEALLOCATE Statement” on page 78) to release space for a pointee array.
- Use the NULLIFY statement (see “NULLIFY Statement” on page 167) to disassociate a pointer variable from a pointee variable.
- Use the POINTER statement (see “POINTER Statement” on page 178) to specify a pointer variable and associate it with a pointee variable.

Arithmetic IF Statement

See “IF Statements” on page 138.

ASSIGN Statement

The ASSIGN statement assigns a number (a *statement label*) to an integer variable. See “Statement Labels” on page 228.

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 to 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 assign-

ment 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:
3 FORMAT (1X,I4)	
ASSIGN 3 TO IFMT	Assign format statement labeled 3 to integer variable IFMT.
ASSIGN 10 TO LABEL	Assign statement labeled 10 to integer variable LABEL.
13 GO TO LABEL,(25,10)	On first execution transfer to statement labeled 10. On second execution transfer to statement labeled 25.
CONTINUE	
10 NUM = 50	
ASSIGN 25 TO LABEL	
GO TO 13	
25 WRITE(5,IFMT) NUM	Write using the FORMAT statement at statement 3.
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, but will not be flagged or diagnosed by the compiler. 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 135.

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 58 shows the rules for conversion in arithmetic assignment statements in the format $a=b$, where the type of b is byte, unsigned, integer, real or complex.

The correspondence between type declarations and data item lengths in bytes is described in Figure 22 on page 102.

Figure 21 (Page 1 of 2). Conversion Rules for the Arithmetic Assignment Statement $a=b$.

Type of a	Type of b							
	BYTE INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER	UNSIGNED	REAL*4 REAL	REAL*8 Double Precision	REAL*16	COMPLEX*8 COMPLEX	COMPLEX*16	COMPLEX*32
BYTE INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER	Assign	Positive assign	Fix and assign	Fix and assign	Fix and assign	Fix and assign real part; imagi- nary part not used	Fix and assign real part; imagi- nary part not used	Fix and assign real part; imagi- nary part not used
UNSIGNED	Drop sign and assign	Assign	Drop sign, fix and assign	Drop sign, fix and assign	Drop sign, fix and assign	Drop sign, fix and assign real part; imaginary part not used	Drop sign, fix and assign real part; imaginary part not used	Drop sign, fix and assign real part; imaginary part not used
REAL*4 REAL	Float and assign	I*4 extend, float, and assign	Assign	Real assign	Real assign	Assign real part; imaginary part not used	Real assign real part; imagi- nary part not used	Real assign real part; imagi- nary part not used
REAL*8 Double Precision	DP float and assign	I*4 extend, 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	I*4 extend, 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	I*4 extend, 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 imagi- nary parts	Real assign real and imagi- nary parts

Figure 21 (Page 2 of 2). Conversion Rules for the Arithmetic Assignment Statement $a=b$.

Type of a		Type of b							
		BYTE INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER	UNSIGNED	REAL*4 REAL	REAL*8 Double Precision	REAL*16	COMPLEX*8 COMPLEX	COMPLEX*16	COMPLEX*32
Type of a	COMPLEX*16 DOUBLE COMPLEX	DP float and assign to real part; imaginary part set to zero	I*4 extend, 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	I*4 extend, 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; imagi- nary part set to zero	QP extend and assign real and imaginary parts	QP extend and assign real and imaginary parts	Assign

Terms in Figure 21 on page 58 are defined as follows:

Assign	Transmit the expression value without change. If the expression value contains more significant digits than the variable <i>a</i> can hold, the value assigned to <i>a</i> is unpredictable.
Positive assign	Transmit the expression value with a positive sign, extending with zeroes on the left, if necessary.
Drop sign	Transmit the expression value without a sign. If the expression value contains more significant digits than the variable <i>a</i> can hold, the value assigned to <i>a</i> is unpredictable.
Real assign	Transmit to <i>a</i> 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.
I*4 extend	Extend the unsigned value to a positive INTEGER*4 value.
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 utility subroutine. See "ASSIGNM Subroutine" on page 340 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
ONEB	Byte	1
NOSIGN1	Unsigned	1
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.
NOSIGN1 = K	The value of NOSIGN1 is replaced by the truncated and signless value of K.
I = ONEB	The value of I is replaced by the sign-extended value of ONEB
I = I + 1	The value of I is replaced by the value of I + 1.
C = NOSIGN1 + D	The value of NOSIGN1 is converted to a positive INTEGER*4 value, then to a REAL*8 value. The sum of this value and D replaces the value of C.
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 6, "Intrinsic Functions" on page 280.
F(1) = A	The value of element 1 of array F is replaced by the value of A.

Statement	Description
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 'ABC' because CHAR1 is of length 10, and the constant is only of length 3; thus CHAR1 is padded with blanks.
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 is part of the static debug which is not recommended for use. Unless you are working with an older version of VS FORTRAN, you should use interactive debug. For more information, see the *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

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 can be many debug packets for one program or subprogram.

Note: The AT statement cannot be used with extended common. For more information on extended common, see the *VS FORTRAN Version 2 Programming Guide*.

For more on debug packets and for examples of the AT statement, see “DEBUG Statement” on page 79.

AUTOMATIC Statement

The AUTOMATIC statement allows you to name the variables and arrays to be in the automatic storage class.

Syntax

```
AUTOMATIC name1 [, name2 ... ]
```

name

is a variable or array. It has the form:

a[(*dim*)]

where:

a is a variable or array

dim

dim is a array declarator. See “Size and Type Declaration of an Array” on page 27 for a full description of *dim*

One type of storage class is automatic. For each data object that is declared automatic, there is exactly one copy for each invocation of a procedure.

The same name must not appear in more than one AUTOMATIC or STATIC statement. Dummy argument names, procedure names, names of constants, and names of pointees must not appear in AUTOMATIC statements. Automatic variables and arrays cannot be initialized (either with a DATA statement or with an explicit type statement). Names declared as AUTOMATIC also cannot appear in COMMON (as an item), SAVE, EXTERNAL, or INTRINSIC statements. Any appearance of *name* in an explicit type statement must precede the AUTOMATIC statement.

Local variables and arrays are static by default.

Cross-Platform Considerations: The AUTOMATIC statement is supported for the ease of migrating source between platforms. Under MVS and CMS all storage is static. However, programs that properly initialize data before it is used should demonstrate the same semantics on MVS and VM as on platforms that use a stack mechanism to support a true automatic class.

BACKSPACE Statement

For sequential access, 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.

For keyed sequential access, 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.

Note: The BACKSPACE statement cannot be used with striped files. The BACKSPACE statement can be used only on a PDS member being processed for input. If the BACKSPACE statement is used with a PDS member, the default BUFNO=1

should be used in the DD statement. A program should not attempt to use the BACKSPACE statement on a PDS member once end-of-data has been reached. For further information on striped files, see the *VS FORTRAN Version 2 Programming Guide*.

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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. 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.

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 or NAMELIST formatting. If either is used, the result is unpredictable.

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 sequential 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.

A BACKSPACE statement should not be issued against a concatenated data set if reading has occurred past the first data set and more than one buffer has been specified.

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.

BLOCK DATA

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 be 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 required.

A particular common block may not be specified 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 138.

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 can 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 can be a variable, array element, or array name; a constant; an arithmetic or logical expression, or an asterisk (*) followed by the statement label of an executable statement that appears in the same program unit as the CALL statement. *arg* can also be a function or subroutine name, or character expression, unless the CALL statement appears within a parallel loop or parallel section.

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, with two exceptions:

When the CALL appears within a parallel loop or parallel section, certain arguments are passed by copy. (When an argument is passed by copy, the value of the argument, at the point of CALL or function reference, is made available to the subprogram. In addition, no alterations to the subprogram's copy are reflected in that of the referencing program during the subprogram execution or on return from the subprogram.) Those passed by copy are expressions (including parenthesized primaries) and loop-control variables of all containing DO and PARALLEL DO loops. (See "Common Blocks in Parallel Programs" on page 74.)

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 103.)

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 26.

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)
```

Example 7:

Assume the following C function is to be called from Fortran code:

```
VOID SORT( INT, INT *, INT * );
```

With the following declarations:

```
INTEGER*4 SIZE
PARAMETER(SIZE=500)
INTEGER*4 VALUES(SIZE), PERMS(500)
PRAGMA ALIAS(SORT, 'SORT')
```


Example of calling a C function from Fortran:

```
CALL SORT(%VAL(SIZE),VALUES,PERMS)
```

Example 8:

Assume the following subroutine definition was made:

```
SUBROUTINE SUBP(I,A,B,C,Z1,Z2)
REAL*4 A(1000),B(1000),C(1000),Z1, Z2, Z3
  :
RETURN
END
```

Example of parenthesized argument in parallel code:

```
PARALLEL DO I=1,1000
LOCAL Z1, Z2, Z3
  :
CALL SUBP(I,A,B,C,(Z1),Z2)
Z1 = Z1 + 4.5
ENDDO
```

I and Z1 will be passed by copy; all other arguments will be passed by reference.

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 99.

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 169.

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.

In an originated task, the file will be deleted for all unnamed files other than the standard input/output units regardless of the value of the STATUS specifier.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes.

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 168.)

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')
```

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, and they cannot be intrinsic function names.

The form // (with no characters or with only blanks between the slashes) denotes blank common. If *name1* 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 or pointer variables. 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 can also be declared for that array. (See “Size and Type Declaration of an Array” on page 27.)

A given common block name can 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 can 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 can 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

```
COMMON CV
COMPLEX*8 CV
.
.
CALL SUB
.
.
STOP
END
```

Subroutine

```
SUBROUTINE SUB
COMMON RV(2)
.
.
RV(2) = 1.2
.
.
RETURN
END
```

Common Blocks in Parallel Programs

Each parallel task of a parallel program has its own copy of blank and named commons. The SCHEDULE statement can be used to specify the sharing of commons by multiple parallel tasks.

Within a parallel task, multiple copies of a common block are created when the common block was not specified in the program unit containing a PARALLEL CALL, or a CALL or function reference within a parallel loop or parallel section, but was specified in the activated subprogram.

In the following example, each iteration of the PARALLEL DO loop has its own copy of common WORK; all iterations will share common A.

```

PROGRAM MAIN
COMMON /A/ A(100)
.
.
PARALLEL DO 10 I=1,100
CALL SUBX(I)
.
.
10 CONTINUE
.
END
SUBROUTINE SUBX(INDEX)
COMMON /A/ A(100)
COMMON /WORK/ B(100,100)
DO 20 J=1,100
DO 30 K=1,100
.
.
B(J,K) = FN(...)
30 CONTINUE
20 CONTINUE
.
.
A(INDEX) = SUM
RETURN
END

```

Blank and Named Common Blocks

Variables and arrays can 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 can be mixed character and noncharacter 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 can be many *named* common blocks, each with its own name.

- *Blank* common blocks can 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 can 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 99.

Computed GO TO Statement

See “GO TO Statements” on page 135.

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 can be placed anywhere in the source program (where an executable statement can appear) without affecting the sequence of execution. It can 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 89.) 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 can, optionally, be signed. Any of these constants can 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.

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 .

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

If a *boz* 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 *boz* 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.

Dummy arguments; names of constants, functions, and statement functions; pointees; pointers; and objects in the automatic storage class must not be assigned initial values.

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/

```

DEALLOCATE Statement

The DEALLOCATE statement releases space used for a pointee array, and resets the associated pointer variable to the unassigned state.

Syntax

```
DEALLOCATE (pointee1 [, pointee2 ...] STAT=stat)
```

pointee

is the name of the pointee array.

stat

is an INTEGER*4 variable that receives a status code. Possible status codes are:

- 0** The deallocation request was successfully completed.
- 4** One or more pointee arrays were not deallocated, because storage for them was not previously assigned with an ALLOCATE statement. The pointer variables are reset to the unassigned state, but no storage is released.

If the DEALLOCATE statement refers to a dynamically dimensioned array, the DDIM and NODDIM compiler options are ignored; the storage released is always the storage previously allocated by an ALLOCATE statement.

When deallocating storage, be careful that more than one pointer variable does not refer to the same storage space. The pointer variables that are not reset to the

unassigned state will continue to reference storage which is now unassigned. The program might fail. You can use the NULLIFY statement to set the other pointer variables to the unassigned state.

Example:

```
Pointer (Cpt, C(10,10)), (Dpt, D(2,2))
Allocate (C, D, Stat=ir)
If (ir /= 0) Stop 3
Call Addem (C, D)
Deallocate (C, D, stat=ir)
```

Related Statements

- Use the ALLOCATE statement (see “ALLOCATE Statement” on page 54) to assign space for a pointee array.
- Use the NULLIFY statement (see “NULLIFY Statement” on page 167) to disassociate a pointer variable from a pointee variable.
- Use the POINTER statement (see “POINTER Statement” on page 178) to specify a pointer variable and associate it with a pointee variable.

DEBUG Statement

The DEBUG statement is part of static debug which is not recommended for use. Debug instructions can be used only in a debug packet.

Unless you are working with an older version of VS FORTRAN, you should use interactive debug. For more information, see the *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

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.

Note: The DEBUG statement cannot be used with extended common. For more information on extended common, see the *VS FORTRAN Version 2 Programming Guide*.

Syntax

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

An *option* can 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 also appear in the first debug packet in which tracing is desired.

INIT (*i*1, *i*2, ...)

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 can be 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 statements
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.”

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 can be returned from a packet to any point in the program being debugged. In addition, no debug packet can refer to a label defined in another debug packet. A debug packet can 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 multidimensional 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
```

The following 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. A positive value for *ios* signifies that an error has been detected; otherwise, *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. 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.

(For rules governing dimension bounds, see “Size and Type Declaration of an Array” on page 27.)

The appearance of *a* in a DIMENSION statement declares *a* to be an array and provides its array declarator.

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,*)
```

```
CHARACTER*10 string1
```

```
CHARACTER string2
```

```
DIMENSION string2(LEN(string1))
```

DISPLAY Statement

The DISPLAY statement is part of static debug, which is not recommended for use. Unless you are working with an older version of VS FORTRAN, you should use interactive debug. For more information, see the *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

Note: The DISPLAY statement cannot be used with extended common. For more information on extended common, see the VS FORTRAN Version 2 Programming Guide.

The DISPLAY statement displays data in NAMELIST output format. It can 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 must not appear in the list.

For examples and explanations of the DISPLAY statement, see “DEBUG Statement” on page 79.

DO and DO WHILE Statements

The DO and DO WHILE statements control the execution of the statements comprising the loop.

Syntax

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

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

stl is the statement label of the **terminal statement**.

i is the name of an integer or real variable called the **DO variable**. Except by incrementation processing, the DO variable can neither be redefined nor become undefined during execution of the range of the DO loop.

Parallel Considerations: When a DO loop is nested within a parallel loop or section, the DO variable automatically becomes a private variable within the range of the containing parallel loop or section. As private variables, these DO variables are undefined upon exit from the parallel loop or section.

e1

e2

e3 are integer or real expressions that specify the initial, test, and incrementation values, respectively.

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

logical_expr

is a logical expression.

The DO or DO WHILE statement, the terminal statement, and the intervening executable statements, specify a loop, called a **DO loop** or a **DO WHILE loop**, respectively.

The terminal statement is the executable statement at the end of the DO or DO WHILE loop. If *stl* is not specified, the terminal statement must be an END DO statement. The terminal statement must follow the DO or DO WHILE statement and is any action statement except: unconditional GO TO, assigned GO TO, arithmetic IF, RETURN, STOP, END. DO and DO WHILE loops can share a labeled terminal statement.

Range of a DO or DO WHILE Loop

The **range of a DO loop** consists of all the executable statements following the DO or DO WHILE statement, up to and including the terminal statement.

Control must not be transferred into the range of a loop from outside the loop.

If a DO or DO WHILE statement appears within the range of another DO loop (called the outer loop), the DO or DO WHILE loop that statement begins (called the inner loop) must be entirely within the range of the outer loop. If a DO or DO WHILE statement appears within a statement block of an IF construct, the loop must be entirely within that statement block.

Active and Inactive DO and DO WHILE Loops

A loop is either active or inactive. Initially inactive, a loop becomes active when the DO or DO WHILE statement is executed. Once active, the loop becomes inactive when:

- In the case of a DO, its iteration count becomes zero
- In the case of a DO WHILE, *logical_expr* evaluates to false
- A RETURN statement is executed within the range of the loop
- Control is transferred to a statement in the same program unit but outside the range of the loop (with a GO TO or alternate return from a subroutine, for example)
- Execution is terminated (with a STOP, for example)

When a DO loop becomes inactive, the DO variable keeps the last value assigned to it.

Execution of DO and DO WHILE Loops

When the DO or DO WHILE statement is executed, the loop is initialized:

- For a DO loop: the initial parameter m_1 , the test parameter m_2 , and the incrementation parameter m_3 are established by evaluating $e1$, $e2$, and $e3$, respectively. Evaluation includes, if necessary, conversion to the type of the DO variable. m_3 must not have a value of zero. If $e3$ is not specified, the default is 1. The DO variable is assigned m_1 and the **iteration count** is assigned

$$\text{MAX} (\text{INT} ((m_2 - m_1 + m_3) / m_3), 0)$$

Note that the iteration count is 0 whenever:

$$m_1 > m_2 \text{ and } m_3 > 0, \text{ or} \\ m_1 < m_2 \text{ and } m_3 < 0$$

- For a DO WHILE loop, there is no initialization.

Loop control processing determines if further execution of the range of the loop is required. For a DO loop, further execution is required if the iteration count is not zero. For a DO WHILE loop, *logical_expr* is evaluated, and if true, further execution is required.

If further execution is not required, the loop becomes inactive. If the loop has a shared terminal statement with one or more loops that are active, execution continues with the innermost active loop, just as if the terminal statement for that loop had just been executed. Otherwise, the normal execution sequence, following the terminal statement, continues.

If further execution is required, execution of the range of the loop begins with the first statement in that range, and continues within that range as determined by the normal execution sequence and transfers of control until either the terminal statement is executed or the loop becomes inactive.

If the terminal statement is executed and the DO loop remains active, then:

- For a DO loop, incrementation processing increments the value of the DO variable by m_3 and decrements the iteration count by 1. Execution of the loop then continues with loop control processing.
- For a DO WHILE loop, execution of the loop continues with loop control processing.

Examples of DO Loops

```
do 20 i = 2, 5
    earliest(i) = 0.0
    do 10 j = 1, i-1
        if (network(j,i) .ne. 0.0)
            x    earliest(i) = max(network(j,i)+earliest(j), earliest(i))
    10    continue
20    continue
```

```

converge = .false.
do while (.not. converge)
  call fitter (x, y, z, converge)
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*4 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 can 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

The implied DO list in an input/output statement is:

Syntax

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

where:

dlist

is an input/output list.

i is the name of an integer (other than Integer*8), 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.

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.

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.

When an implied DO does not reach normal completion due to an "END=", "ERR=", or "NOTFOUND=" branch being taken, the value of the implied DO variable becomes undefined.

DOUBLE COMPLEX Type Statement

See "Explicit Type Statement" on page 99.

DOUBLE PRECISION Type Statement

See "Explicit Type Statement" on page 99.

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 138.

ELSE IF Statement

See "IF Statements" on page 138.

END Statement

The END statement is used as a delimiter to define a program unit. It terminates a main program, or a function, subroutine, or block data subprogram.

Syntax

```
END
```

The END statement can be numbered. It cannot be continued, and no other statement in the program unit can have an initial line that appears to be an END statement. An END statement in a main program terminates the running of the program. An END statement in a function or subroutine subprogram has the same effect as a RETURN statement.

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

ENDFILE

END Statement in a Subprogram

All function subprograms must end with END statements, and 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 is part of the static debug which is not recommended for use. Unless you are working with an older version of VS FORTRAN, you should use interactive debug. For more information, see the *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

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

Note: The END DEBUG statement cannot be used with extended common. For more information on extended common, see the *VS FORTRAN Version 2 Programming Guide*.

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 79.

END DO Statement

The END DO statement terminates the range of a DO loop, PARALLEL DO loop, or DO WHILE loop.

Syntax

END DO

The END DO statement must be used if the DO , PARALLEL DO, or DO WHILE statement that specifies the 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 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. A positive value for *ios* signifies that an error has been detected; otherwise, *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes.

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.

When processing partitioned data sets, after a member is written, a CLOSE or REWIND statement (not an ENDFILE statement) must be specified for the unit representing the member before another member is written. This closes the partitioned data set after each member is created so that the end-of-file (EOF) record is supplied correctly for that member.

END IF Statement

See “IF Statements” on page 138.

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 50.) 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 27.)

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 $\text{SQRT}(X^{**}J)$ and returns the value in Z.

The subprogram invocation

```
CALL ALIAS(Y,W)
```

evaluates the expression $\text{SQRT}(Y)$ 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” on page 230 and “Actual Arguments in a Function” on page 133.

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 they refer.

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” on page 231 and “Dummy Arguments in a Function” on page 133.

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 can be integer expressions containing only integer constants, or names of integer constants. They must not contain variables, array elements, or function references.

The integer expressions can contain the character intrinsic functions ICHAR, and LEN. The arguments to ICHAR and LEN must be constants. The argument to LEN must not be a dummy argument that has been specified with an inherited length.

All the named data within a single set of parentheses shares the same storage location.

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, therefore entities in automatic storage class cannot be equivalenced to those in static storage class.

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

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 it is in the static storage class. 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 can 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 cannot 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 default implied types.

Syntax

```
type name1 [, name2 ... ]
```

type

is COMPLEX, DOUBLE COMPLEX, INTEGER, LOGICAL, REAL, DOUBLE PRECISION, CHARACTER[**len*[,]], BYTE, or UNSIGNED

where:

len

specifies the length (number of characters between 1 and 32767). If specified, it is assumed to be 1.

Note: The CHARLEN compile-time option can 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 expression can contain the character intrinsic functions ICHAR and LEN. The arguments to ICHAR and LEN must be constants. The argument to LEN must not be a dummy argument that has been specified with an inherited length.

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.

The comma in character[**len*[,] must not appear if **len* is not specified.

type

is COMPLEX[**len*1], INTEGER[**len*1], LOGICAL[**len*1] REAL[**len*1], or UNSIGNED[**len*1]

where:

**len*1

*len*1 represents one of the permissible length specifications for its associated type as described in Figure 11 on page 24.

name

is a variable, array, function name, or dummy procedure name, or the name of a constant. It has the form:

a[(*dim*)][**len2*]

where:

a is a variable, array, function name, or dummy procedure name.

dim

dim can be specified only for arrays. It is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array.

(For rules about dimension bounds, see “Size and Type Declaration of an Array” on page 27.)

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 6, “Intrinsic Functions” on page 280, 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 can 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 131).

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

dim can be specified only for arrays. See the description of *dim* above.

i1,i2,i3,...

are optional and represent initial data values.

Initial data values can be assigned to variables or arrays 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 to assign initial values only to array elements. The data must be of the same type as the variable or array, except that *boz* data can be used for any type.

Successive occurrences of the same constant can be represented by the form *r*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 type statement or in a preceding specification statement.

Initialization by use of *in* may also occur on the DOUBLE PRECISION, DOUBLE COMPLEX, and BYTE type statements.

For more information on initial values and allowable objects, see “DATA Statement” on page 76.

Figure 22 lists all the possible explicit type statements, and the resulting type and length of the data item.

Figure 22. Type and Length of Explicit Type Statements

Type Statement	Resulting Type	Length (Bytes)
BYTE	BYTE	1
CHARACTER	CHARACTER	1
CHARACTER*n	CHARACTER	n (where $1 \leq n \leq x$) See note.
COMPLEX	COMPLEX	8
COMPLEX*8	COMPLEX	8
COMPLEX*16	COMPLEX	16
COMPLEX*32	COMPLEX	32
DOUBLE COMPLEX	COMPLEX	16
DOUBLE PRECISION	REAL	8
INTEGER	INTEGER	4
INTEGER*1 INTEGER*2	INTEGER	1 2
INTEGER*4	INTEGER	4
INTEGER*8	INTEGER	8
LOGICAL	LOGICAL	4
LOGICAL*1	LOGICAL	1
LOGICAL*2	LOGICAL	2
LOGICAL*4	LOGICAL	4
LOGICAL*8	LOGICAL	8
REAL	REAL	4
REAL*4	REAL	4
REAL*8	REAL	8
REAL*16	REAL	16
UNSIGNED	UNSIGNED	1
POINTER*4	POINTER	4
POINTER*8	POINTER	8

Note: If the CHARLEN compile-time option is not specified, $x=500$ (or whatever length was specified as the default at installation). If CHARLEN is specified, $x=\text{CHARLEN}$, 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*10 string1
```

```
CHARACTER*(LEN(string1)) string2
```



```

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 function subprogram, function or subroutine subprogram 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 VS FORTRAN-supplied 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 in the same program unit. Instead, it is assumed that the function is supplied by the user.

However, when the same name is used as a user-supplied function in one program unit and as an intrinsic function in another program unit in the same load module, the user has the responsibility of ensuring proper selection of the user-supplied routine.

There are two ways you can ensure proper selection of the user-supplied routine. You can define the partitioned data set containing the load module of the external function with a ddname other than SYSLIB and use the following statements. The example uses the ddname MYLIB.

```
//SYSLIN DD DSN=OBJECT-OF-MAIN,...
          DD *
          INCLUDE MYLIB (MEMBERNAME-OF-EXTERNAL-FUNCTION)
```

You can concatenate the partitioned data set containing the load module of the external function before the VS FORTRAN libraries in the SYSLIB dd statement and use the following statements:

```
//SYSLIN DD DSN=OBJECT-OF-MAIN, ...
//      DD *
          INCLUDE SYSLIB(MEMBERNAME-OF-EXTERNAL-FUNCTION)
```

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. Figure 23 lists the valid format codes:

Figure 23 (Page 1 of 2). Format Code Descriptions

Code	Format	Description
I	<i>alw</i>	Integer or unsigned data fields
I	<i>alw.m</i>	Integer or unsigned data fields
D	<i>aDw.d</i>	Double precision data fields
E	<i>aEw.d</i>	Real data fields
E	<i>aEw.dEe</i>	Real data fields
F	<i>aFw.d</i>	Real data fields
G	<i>aGw.d</i>	Real data fields
G	<i>aGw.dEe</i>	Real data fields
P	<i>nP</i>	Scale factor
L	<i>aLw</i>	Logical data fields
A	<i>aA</i>	Character data fields
A	<i>aAw</i>	Character data fields
	'character constant' "character constant"	Literal data (character constant)
H	<i>wH</i>	Literal data (Hollerith constant)
X	<i>wX</i>	Input: Skip a field Output: Fill with blanks
T	<i>Tr</i>	Transfer of data starts in current position
TL	<i>TLr</i>	Transfer of data starts <i>r</i> characters to the left of current position
TR	<i>TRr</i>	Transfer of data starts <i>r</i> characters to the right of current position
group	<i>a(...)</i>	Group format specification
S	<i>S</i>	Display of optional plus sign is restored
SP	<i>SP</i>	Plus sign is produced in output
SS	<i>SS</i>	Plus sign is not produced in output
BN	<i>BN</i>	Blanks are ignored in input
BZ	<i>BZ</i>	Blanks are treated as zeros in input
slash	<i>/</i>	Data transfer on the current record is ended
dollar	<i>\$</i>	End-of-record is inhibited for formatted write statements to external sequential files

Figure 23 (Page 2 of 2). Format Code Descriptions

Code	Format	Description
colon	:	Format control is terminated if there are no more items in the input/output list
E	<i>aEw.dDe</i>	Extended precision data fields
G	<i>aGw.d</i>	Integer or logical data fields
G	<i>aGw.dEe</i>	Integer or logical data fields
Q	<i>aQw.d</i>	Extended precision data fields
B	<i>aBw[m]</i>	Binary data fields
O	<i>aOw[m]</i>	Octal data fields
Z	<i>aZw[m]</i>	Hexadecimal data fields
Note: <i>a</i> 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 <i>a</i> is 1 to 255. If <i>a</i> is omitted, the code or group is used only once. <i>w</i> is an unsigned, nonzero, integer constant that specifies the width of the field. This width must be less than 256. <i>m</i> is an unsigned integer constant that specifies the number of digits to be printed. <i>d</i> is an unsigned integer constant that specifies the number of digits to the right of the decimal point. <i>e</i> is an unsigned, nonzero, integer constant that specifies the number of digits in the exponent field. <i>n</i> is an (optionally) signed integer constant that specifies a scale factor to be applied. The range is $-127 \leq n \leq 127$. <i>r</i> is an unsigned, nonzero, integer constant that specifies a character position in a record. <i>(...)</i> 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. The FORMAT statement describes the data fields with format codes. The order of these format codes determines the structure of the Fortran records. The READ or WRITE I/O list gives the names of the data items that make up the record. Together, the length of the list and the FORMAT statement codes specify the length of the record. (See “Forms of a FORMAT Statement” on page 109.)

The format codes delimited by left and right parentheses can 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 ** 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 can 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 can be different and the format codes T, TL, TR, X, /, :, S, SP, SS, P, BN, BZ, H, \$, or a character constant can appear between the two codes.

- When a FORMAT defines a record, 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 carriage 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. 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; however, only characters blank and zero 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 121). Writing to a variable format file causes an error.

To produce a blank record on output, an empty format specification of the form FORMAT () can 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 176).
- If you use the comma as an input delimiter with the format codes I, D, E, F, G, L, Q, B, O, and Z, you do not need to align the data with leading zeroes or blanks. If a comma or end of the record lies within the range of the format field when dealing with non-character input, the field width is reduced to the size up to, but not including, the comma and up to the end of the record. The next field begins immediately after the comma or after the end of the record. If additional data is required from the record when the current field pointer is outside the record, then an error message is issued. 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 B, I, D, E, F, G, O, Q, B, O, 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          (a comma occurs immediately after the previous field)
II = 32
X = .00001       (not 1.00000, since no decimal was used in the input)
```

And the ,7 would be ignored.

In another example, using the same program fragment and the following input:

```
1,abcdefgh 32,,1,7
```

The following assignments would occur:

```
A = .01
C = abcdefgh
B = .00032
II = 0          (because of the two consecutive commas)
X = .00001
```

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 168) and any BN or BZ blank control that is currently in effect. Plus signs can 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 can 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 can 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 Ew.dEe or Gw.dEe 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 `Ew.dDe` or `Qw.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 `/` format code
- Before or after a `:` format code
- Before or after a `$` 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`, `B`, `O`, 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', "character constants", `S`, `SP`, `SS`, `BN`, `BZ`, `P`, the slash (`/`), the colon (`:`), or the dollar (`$`). These communicate information directly to the record.

Whenever an `I`, `D`, `E`, `F`, `A`, `G`, `L`, `Q`, `B`, `O`, 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, B, O, 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', "character constants", colons, slashes, and dollars 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 106. If the form *Iw.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 the number is greater than *w*, asterisks are printed instead of the number. If the form *Iw.m* is used, the output is the same as the *Iw* 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 *Fw.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 can have more digits than VS FORTRAN Version 2 uses to approximate the value of the datum. The basic form can 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 $Dw.d$, $Ew.d$, $Ew.dEe$ format codes edit real, complex, or double precision data.

The $Ew.dDe$ and $Qw.d$ 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 can have more digits than VS FORTRAN Version 2 uses to approximate the value of the datum.

Input datum must be a number, that, optionally, can have a D, E, or Q exponent, or that can 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 106.

The input datum can 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 $Gw.d$ and $Gw.dEe$ 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 $N \geq 10^{**d}$

(where k is the scale factor currently in effect), then:

- $Gw.d$ output editing is the same as $kPEw.d$ output editing.
- $Gw.dEe$ output editing is the same as $kPEw.dEe$ 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 shown in Figure 24.

Figure 24. FORMAT Scale Factor

Magnitude of Data	Equivalent Conversion
$0.1 \leq N < 1$	$F(w-n).d, n ('')$
$1 \leq N < 10$	$F(w-n).(d-1), n ('')$ $F(w-n).1, n ('')$
$10^{**}(d-2) \leq N < 10^{**}(d-1)$	$F(w-n).0, n ('')$
$10^{**}(d-1) \leq N < 10^{**}d$	

n means:

- 4 for $Gw.d$
- $e+2$ for $Gw.dEe$

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 can 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 can, but need not, be placed after the P format code—for example, 2P,3F7.4 or 2P3F7.4 - when a repeat count is specified.) A scale factor of zero can 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

that can 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

that can also be written -2P,F7.4.

Output: With an F format code, the internally represented number multiplied 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

that can 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

$$.1 \leq m < 10^{**d}$$

where d is the number of digits as specified in the G format code Gw.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

B Format Code

The Bw.m format code *transmits binary data*.

Input: Scanning of the input field proceeds from right to left. The input value must consist only of the characters 0 and 1. One bit in internal storage contains one binary digit. If the storage area is too small for the input data, the data is truncated and high-order digits are lost.

Examples

Format Specification	Input Value	Result Value
B6	101100	44
B8.3	01100111	103

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. If the form Bw.m is used, the output is the same as the Bw form, except that the 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.

Examples

Format Specification	Source Value	Output Value
B8	19	00010011
B4	19	0011
B8.6	19	010011

O Format Code

The Ow.m format code *transmits octal data*.

Input: Scanning of the input field proceeds from right to left. If the storage area is too small for the input data, the data is truncated and high-order digits are lost.

Examples

Format Specification	Input Value	Result Value
O3	100	64
O4.2	1032	282

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. If the form Ow.m is used, the output is the same as the Ow form, except that the 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.

Examples

Format Specification	Source Value	Output Value
O5	1039	02017
O4	19	0011
O8.6	19	010011

Z Format Code

The Zw.m 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 hexadecimal values:

```

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. If the form Zw.m is used, the output is the same as the Zw form, except that the 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.

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)
<b><b><b><b>3355    (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 109 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.

FORMAT

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
	b3F1156432D+02276.38E+15bbbbbbbbbb				
Input	2AF3155381+02b382506E+28276.38E+15				
Lines	3ACb346.18D-03485.322836276.38E+15				
Format:	Z4	D10.3	G10.3	G10.3	

Then *b* 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:

Figure 25. Lines produced by 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 *edits 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. Lowercase alphabetic data may be folded to uppercase when input is from the terminal.

Issue the command `FILEDEF FT05F001 TERMINAL (LOWCASE)`, to inhibit this on CMS.

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 (*b* represents a blank).

Input Line	B	N	M
ABCDEFG46bATb11234567	ABCDEFGb	ATb1	4567
HIJKLMN76543213334445	HIJKLMNb	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 to 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 (*b* represents a blank):

A	B	C	Printed Line
A1B2	C3D4	E5F6G7H8	bbA1B2bC3D4E5F6G7

H Format Code and Character Constants

Character constants can appear in a FORMAT statement in one of three ways: following the H format code, enclosed in apostrophes, or enclosed in quotation marks. For example, the following FORMAT statements are equivalent.

```
25  FORMAT (22H 1982 INVENTORY REPORT)

25  FORMAT (' 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 or quotation marks are used, the characters enclosed in apostrophes or quotation marks 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 carriage control character for printed output. One line is skipped before printing, and the carriage 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. If the character constant is enclosed in quotation marks, a quotation mark in the data is represented by two successive quotation marks. For example, DON'T would be represented as 'DON''T'; and SAY "THANK YOU" TO THE NICE OPERATOR would be represented as "SAY ""THANK YOU"" TO THE NICE OPERATOR". The two successive apostrophes or quotation marks 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 carriage 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 carriage 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 109.) 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 168.)

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 nonleading blank characters in succeeding numeric fields are treated as zeros. If no OPEN statement is given and the file is preconnected, all nonleading 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 (/) format code *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.

If a / format control code is encountered after the \$ format control code, the / code is honored and the current record is ended.

Dollar (\$) Format Code

A dollar (\$) format code *inhibits the end-of-record* for formatted WRITE statements to sequential files. This is especially useful for files written to the terminal.

Note: Due to the singular characteristics of terminal I/O under CMS, \$ format control may have no discernible effect; output appears within the terminal output area, while input is entered in the terminal input area.

The \$ format code has no effect on WRITE statements to internal files, direct access files, or keyed access files, or on READ statements.

When the \$ format code is encountered in a FORMAT specification, the normal end-of-record condition at the end of the FORMAT specification is suppressed. A subsequent WRITE statement will add data to the current record.

If a / format control code is encountered after the \$ format control code, the / code is honored and the current record is ended.

Colon Format Code

A colon (:) format code *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
      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 can then be used as the format specification for subsequent input/output operations. The format specification can also be placed in 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, or two consecutive quotation marks within a character field that is defined by quotation marks, it should be used for output only.
- Blank characters can precede the format specification, and character data can 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 can 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 can 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 or quotation marks. Each apostrophe or quotation mark within a character constant delimited by apostrophes or quotation marks, respectively, must be represented by two consecutive apostrophes or quotation marks without an intervening blank or the end of the record. Character constants can 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 can be continued on as many records as needed. The characters blank, apostrophe, quotation mark, comma, and slash can 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 can 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 `Iw` 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:

$$10^{**d1} \leq x < 10^{**d2}$$

where $d1$ and $d2$ are processor-dependent integer values. If the magnitude x is within this range, the constant is produced using `0PFw.d`; otherwise, `1PEw.dEe` 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 can 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 or quotation marks by default. The `DELIM=` specifier of the `OPEN` statement can be used to override this default.
- Have each internal apostrophe or quotation mark represented externally by two apostrophes or quotation marks only if the `DELIM=` specifier in effect is "apostrophe" or "quote", respectively
- Are not preceded or followed by a value separator
- 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 carriage 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 can include one or more RETURN statements. The identified function subprogram is an independently written program to which control will be transferred wherever its name is referred to in another program.

Syntax

```
[type] FUNCTION name ( [ arg1 [, arg2... ] ] )
```

type

is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, LOGICAL, CHARACTER[**len*] or UNSIGNED

where:

**len*

is the length specification, as given under “Explicit Type Statement” on page 99 except arguments to ICHAR and LEN must be constants and the expression 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[**len1*]

is the name of the function.

where:

**len1*

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 23.) It can be included only when *type* is specified as INTEGER, REAL, COMPLEX, LOGICAL, or UNSIGNED. 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 can 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” on page 133.)

A type declaration for a function name can 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. 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 can 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, it must follow the FUNCTION statement and can 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 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, SCHEDULE, or PARALLEL CALL statement that will cause a value to be assigned in the referenced subroutine
- 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 function. For additional information on RETURN and END statements in a function, see “RETURN Statement” on page 217 and “END Statement” on page 91.

The function can also use one or more of its arguments to return values to the calling program.

The dummy arguments of the function (*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 27.

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

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 (not if within parallel loop or section)
- An external procedure name (not if within parallel loop or section)
- 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, see “ENTRY Statement” on page 94.

Dummy Arguments in a Function

The dummy arguments of a function 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 should appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, SAVE, INTRINSIC, AUTOMATIC, STATIC, or POINTER statement, except as a common block name, 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 function, 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 SUFFIX:

```
CHARACTER*10 FUNCTION SUFFIX(STR)
CHARACTER*7 STR
SUFFIX = STR // 'SUF'
END
```

Use of function SUFFIX:

```
CHARACTER*10 NAME, SUFFIX
.
.
NAME = SUFFIX(NAME(1:7))
```

2. Definition of function CUBE. This illustrates a function defined without dummy arguments:

```
REAL FUNCTION CUBE*16()
COMMON /COM1/ A
CUBE = A * A * A
END
```

Use of function 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 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 can 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 )

```

Such argument mismatches cannot be detected by the compiler unless the Inter-
compilation Analysis (ICA) facility is used. See the VS FORTRAN Version 2 Pro-
gramming Guide for information on the ICA.

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

These statements should not branch into the range of a loop or a section.

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 55.)

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, (*stl1*, *stl2*, *stl3* ...), is optional. If the statement labels are omitted, the preceding comma must also 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 can 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 cannot 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 can 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 $1 \leq m \leq n$, where *n* is the number of statement labels, the next statement is executed.

A computed GO TO statement can 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 can 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 can 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 ( m1 ) 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 ( m1 ) 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.
- A DISPLAY statement.
- A PARALLEL DO, DOBEFORE, DOEVERY, or DOAFTER statement.
- A PARALLEL SECTIONS, SECTION, or END SECTION statement.
- An END DO 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* can have a statement label. If a logical IF statement terminates the end of a DO loop, it cannot 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. It may also be used to change the default storage class.

Syntax

IMPLICIT *type* (*a* [, *a*] ...) [, *type* (*a* [, *a*] ...)] ...

IMPLICIT NONE

type

is CHARACTER[**len*], COMPLEX, DOUBLE COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, REAL, BYTE, UNSIGNED, COMPLEX*8, COMPLEX*16, COMPLEX*32, INTEGER*1, INTEGER*2, INTEGER*4, INTEGER*8, LOGICAL*1, LOGICAL*2, LOGICAL*4, LOGICAL*8, REAL*4, REAL*8, REAL*16. UNSIGNED*1, AUTOMATIC, or STATIC.

len

specifies the length of a character entity. It is either an unsigned, nonzero, integer constant, or a positive constant expression enclosed in parentheses and it has a positive value. The constant expression can contain the character intrinsic functions ICHAR and LEN. ICHAR and LEN must specify

constant arguments. LEN must not use a dummy argument that has been specified with an inherited length.

- 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, except for those containing AUTOMATIC or STATIC. A PARAMETER statement must 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 that begin with non-EBCDIC double-byte characters. A letter or range of letters specified as STATIC or AUTOMATIC can also appear in an IMPLICIT statement for any data type.

For any name, type specification by an IMPLICIT statement can 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-\$)

IMPLICIT CHARACTER*(ICHAR('9') - ICHAR('a')) (c)

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 Directive

INCLUDE is a compiler directive that is coded in your source program. It causes source from another file to be read and processed. When the end of the INCLUDE file is reached, processing resumes with the line following the INCLUDE directive.

A facility called conditional INCLUDE provides a means for selectively activating INCLUDE directives within the source program during compilation. INCLUDE files can reside on any accessed minidisk on CMS or on a cataloged volume in an MVS system. The included files (the subset) are specified by means of the CI compiler option. For more information about the CI compile-time option, see *VS FORTRAN Version 2 Programming Guide*.

The INCLUDE directive has two syntax forms. The first form lets you specify the name of a CMS or MVS source library member. The second form lets you specify a CMS file or MVS data set.

In the first form, a file definition (FILEDEF command, ALLOCATE command, or DD statement) is required when specifying a source library member.

Syntax 1

INCLUDE (*name*) [*n*]

name

a sequence of 1 to 8 letters or digits, the first of which must be a letter. *name* is the member-name of a source library member that 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 form of the INCLUDE directive conforms to the Systems Application Architecture. The file specifier is in one of the following formats:

- **On CMS:** *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, unless the file is a MACLIB member, in which case MACLIB is the default file type.

filemode

from 1 to 2 characters where the first character ranges from A to Z and the second character ranges from 0 to 6. An asterisk (*) can also be specified for the file mode. If an asterisk is specified, the CMS search

INCLUDE

order is used to locate the first file with the same file name and file type. If *filemode* is omitted, (*) will be used.

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: (1) Lowercase letters are equivalent to uppercase letters even though it is possible to create file names with lowercase letters in CMS. One or more blanks must separate each part of the CMS file identifier. (2) If you use an INCLUDE directive with the filemode number 4 (for example, X'A4'), the program will abend or a diagnostic message will be issued.

- **On MVS:** *dsn [(member)]*

dsn

a fully-qualified MVS data set name (cataloged). 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).

Note: Lowercase letters are equivalent to uppercase letters.

The following rules apply to INCLUDE:

- INCLUDE must not be continued.
- The source inserted by INCLUDE can contain any Fortran source statements or other INCLUDE directives.
- The first noncomment line of the included group of statements must not be a continuation line.
- An INCLUDE of a group must not contain an INCLUDE directive that refers to a currently open INCLUDE group (that is, recursion is not permitted).
- Multiple INCLUDE directives can appear in the original source program.
- INCLUDE directives can appear anywhere in a source program before the END statement, except as the trailer of a logical IF statement. An END statement can be part of the included source.
- The statements 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 the source, the resulting program must follow all rules for sequencing of statements.

Valid INCLUDE Directives:**For Syntax 1**

```
INCLUDE (MYFILE)
```

```
INCLUDE (DATA) 2
```

For Syntax 2

```
INCLUDE 'Fully.Qualifid.Data.Set.Name'      (MVS)
```

```
INCLUDE 'Common.MacLib.Lib(Subr1)'          (MVS)
```

```
INCLUDE 'Common MacLib ( Subr1 )'          (CMS)
```

```
INCLUDE 'Common ( Subr1 )'                 (CMS)
```

```
INCLUDE 'InclSet8 Incl'                    (CMS)
```

INQUIRE Statement

The INQUIRE statement supplies information about a particular file or about a particular external unit. You can use INQUIRE to check particular characteristics of a file so you can take alternative actions based on the information returned.

When you are running a parallel program, an INQUIRE statement processed from within an originated task returns the status of that file within the originated task. For example, if a file is connected in task 1 but not in task 2, an INQUIRE in task 2 will report that the file is not connected.

There are three types of INQUIRE statements:

- INQUIRE by file name
- INQUIRE by unit
- INQUIRE by unnamed file

For a concise summary see Figure 29 on page 160.

INQUIRE by File Name

Allows you to gather information about a named file. If you select this form, specify the file name on the FILE specifier. For a discussion of named and unnamed files, see “Input/Output Semantics” on page 49.

Syntax

```
INQUIRE
  (FILE=fn
   [,specifier-list])
```

FILE=fn

fn is a character expression used to identify a file. It must be preceded by FILE=. It can take one of the following formats:

- **On CMS and MVS:** *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 CMS:** */filename filetype [filemode] [(member)]*

a character expression whose value, when any trailing blanks are removed, is the filename, filetype, and filemode.

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 \$). If filetype is not specified, a filetype of Fortran is assumed, unless the file is a MACLIB member, in which case MACLIB is the default filetype.

filemode

from 1 to 2 characters where the first character ranges from A to Z and the 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 filemode.
- If an asterisk (*) is specified for the filemode, the standard CMS search order will be used to find an existing file; if the file is found and the NAME specifier is used, then INQUIRE will return the actual filemode.

member

a member name, 1 to 8 characters long, and the parentheses are required to indicate that it is a member name. Use *member* only when you refer 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 routine can 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 327.

In addition, the following default CMS file identifiers must not be specified on the FILE specifier:

- FILE FT nn F mmm , 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 FT nn K kk , 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.
- FTSExxxx, FTSPxxxx, FTSQxxxx, FTSRxxxx, FTUExxxx, FTUPxxxx, FTUQxxxx, FTURxxxx

- **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 lowercase 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 327.

- **Other CMS and MVS Considerations:**

If the FILE specifier is omitted, one of the default names for unnamed files is assumed. Default names must not be specified on the FILE specifier. Default names 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 names of this form. In an originated task, unnamed files do not have default ddnames, except for those files connected to the default units, i.e. FTSExxxx, FTUExxxx, etc.
- The ddname for an unnamed file using striped I/O must take the form:

FTnnPmmm

where *nn* is the 2-digit unit number and *mmm* is the 3-digit stripe number. For more information on striped I/O see *VS FORTRAN Version 2 Programming Guide*.

- FTnnKkk, where *nn* is the two-digit unit identifier and *kk* is the two-digit sequence number. Keyed files are assigned default names of this form.
- FTERRsss, where *sss* is the three-digit MTF subtask number. For the error message unit of an MTF subtask, the name defaults to this form.
- FTPRTsss, where *sss* is the three-digit MTF subtask number. For the PRINT/WRITE unit of an MTF subtask, the name defaults to this form. (If the PRINT/WRITE unit and the error message unit are the same, the PRINT/WRITE unit's default name is FTERRsss.)

INQUIRE

- FTSExxxx, FTSPxxxx, FTSQxxxx, FTSRxxxx, FTUExxxx, FTUPxxxx, FTUQxxxx, FTURxxxx

Source files that specify system dependent file names in an INQUIRE statement can be compiled on any of the supported operating systems; however, the program may run on only the operating system for which the filename is valid.

INQUIRE by Unit

Allows you to find out whether a unit exists and whether a file is connected to it. If a file is connected to the unit at the time, you can also find out selective characteristics of the file. For a general discussion of file and unit connection, see “Input/Output Semantics” on page 49.

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 49.

Syntax 1

```
INQUIRE
  ([UNIT=]un,
  FILE=bl
  [,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=*bl*

bl is a character expression whose value is blanks. *bl* can be specified as a character constant, as shown in the following example:

```
INQUIRE (UNIT=un,FILE=' ', ...)
```

The following conditions, in the order they are listed, determine which file is referred to:

1. If the unit is connected to an unnamed file that has been connected by an OPEN, READ, WRITE, PRINT or ENDFILE statement, the characteristics of that file are returned.
2. If the above condition is not met and the INQUIRE is being processed from within an originated task, the characteristics of a dynamically allocated temporary file are returned.
3. If the above condition is not met, and a file definition for the ddname FTnnP001 is in effect, the characteristics of that file are returned.
4. If the above condition is not met, and a file definition for the ddname FTnnK01 is in effect, the characteristics of that file are returned.
5. Otherwise: Under CMS and MVS, the characteristics for the file with the default ddname FTnnF001 are returned.

For Syntax 2:**FILE=*ddn***

ddn is 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
- FTnnkk, 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
- FTSExxxx, FTSPxxxx, FTSQxxxx, FTSRxxxx, FTUExxxx, FTUPxxxx, FTUQxxxx, FTURxxxx

For unnamed files that are dynamically allocated under CMS, you may also code the INQUIRE statement as follows; however, in this case, you can inquire only about file existence:

```
INQUIRE (FILE='/FILE FTnnF001 fm', EXIST=EX)
```

Parallel Considerations: When you are not within a parallel originated task and you execute an INQUIRE by unnamed file with 'FTnnPmmm' given for the FILE specifier, the values for FTnnP001 are returned if the file number *mmm* is within the range 001 to the total number of stripes. If you are within a parallel originated task, an INQUIRE by unnamed file using a ddname of the form 'FTnnFmmm' reports that

the unnamed file FTnnFmmm does not exist and cannot be opened within that originated task.

Optional Specifiers

The optional specifiers are listed in Figure 26 and can be specified in any order. Each specifier can appear only 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 29 on page 160 for details.

ACCESS= <i>acc</i>	KEYED= <i>kyd</i>	OPENED= <i>opn</i>
ACTION= <i>act</i>	KEYEND= <i>ken</i>	PAD= <i>pad</i>
BLANK= <i>blk</i>	KEYID= <i>kid</i>	PASSWORD= <i>pwd</i>
CHAR= <i>chr</i>	KEYLENGTH= <i>kle</i>	POSITION= <i>pos</i>
DELIM= <i>dlim</i>	KEYSTART= <i>kst</i>	READ= <i>ron</i>
DIRECT= <i>dir</i>	LASTKEY= <i>lky</i>	READWRITE= <i>rwr</i>
ERR= <i>stl</i>	LASTRECL= <i>lrl</i>	RECL= <i>rcl</i>
EXIST= <i>exs</i>	NAME= <i>nam</i>	SEQUENTIAL= <i>seq</i>
FORM= <i>frm</i>	NAMED= <i>nmd</i>	UNFORMATTED= <i>unf</i>
FORMATTED= <i>fmt</i>	NEXTREC= <i>nxr</i>	WRITE= <i>wri</i>
IOSTAT= <i>ios</i>	NUMBER= <i>num</i>	

Figure 26. Optional Specifiers on the INQUIRE Statement

ACCESS=*acc*

acc is a character variable or character array element. *acc* is assigned a value that corresponds to the access method being used for the currently connected file: 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 the default name of FTnnFmmm), and
2. The unit is preconnected, and
3. No OPEN, READ, or WRITE statement has been issued for the unit.

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.

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. It allows you to check whether a connected file can contain data in a language, such as Japanese, that has double-byte characters. If the file is connected, *chr* is assigned the value DBCS if CHAR='DBCS' was specified on the OPEN statement. If you coded CHAR='NODBCS', did not code the CHAR specifier, or did not use an OPEN statement, the value NODBCS is returned. If the file is not connected, *chr* will become undefined.

DELIM=*dlim*

dlim is a character variable or character array element indicating how character constants processed by list-directed or NAMELIST processing will appear in the output file. It is assigned one of these values:

APOSTROPHE indicates that character constants are enclosed in apostrophes.

QUOTE indicates that character constants are enclosed in quotation marks.

NONE indicates that character constants are not enclosed in either apostrophes or quotation marks for list-directed output.

NONE will always be returned for a NAMELIST file in the absence of an explicit open. (Note that, for compatibility with previous releases of VS FORTRAN, character constants in NAMELIST output are delimited with apostrophes unless the file is opened with the DELIM specifier having the value NONE or QUOTE.)

UNDEFINED indicates that the file or unit is not connected, or that it is connected, but for nonsequential or unformatted input/output.

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. This specifier always returns NO for a striped file. See Figure 27 for additional information.

Figure 27. Characteristics that Determine Whether a File Can Be Connected for Sequential, Direct, or Keyed Access

Specifier	File Characteristics that Result in YES	File Characteristics that Result in NO
SEQUENTIAL	Non-VSAM file	VSAM key-sequenced data set
	VSAM entry-sequenced data set	
	VSAM relative-record data set	
	Striped files	
DIRECT	Non-VSAM file with fixed unblocked records	Non-VSAM file with other than fixed unblocked records
	VSAM relative-record data set	VSAM entry-sequenced data set
		VSAM key-sequenced data set
		Striped files
		Subfiles with sequence number > 001.
KEYED	VSAM key-sequenced data set	Non-VSAM file
		VSAM entry-sequenced data set
		VSAM relative-record data set
		Striped files
		Subfiles with sequence number > 001.

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*.

EXIST=exs

exs is a logical variable or logical array element of length 4. This specifier allows you to check whether a particular unit or file exists for your program. 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.

If you are using INQUIRE by file name: *exs* is assigned the value true if the file of the specified name exists; otherwise, it is assigned the value false. File existence depends on a number of factors, such as the type of device the file is on. *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. A unit exists if its unit number is within the range of allowable unit numbers at your site.

If you are using INQUIRE by unnamed file: *exs* is assigned the value true if both the unit and the unnamed file exist; otherwise, *exs* is assigned the value false. *exs* becomes undefined if an error occurs. For a striped file, *exs* is assigned the value true if the file specified by FTnnP001 exists; *exs* is assigned

the value false, however, if the stripe number *mmm* is greater than the total number of stripes, even if FTnnP001 exists. For more information see the *VS FORTRAN Version 2 Programming Guide*.

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.

FORMATTED=fmt

fmt is a character variable or a character array element. Based on the file's characteristics, it is assigned a value of YES or NO as shown in Figure 28. It returns UNKNOWN if it is not possible to determine whether the file can be connected for formatted access.

Figure 28. Characteristics That Determine Whether a File Can Be Connected for Formatted or Unformatted I/O

	File Characteristics that Result in YES File Character-Specifier	File Characteristics that Result in NO File Character-Specifier
FORMATTED	Non-VSAM file with other than variable-length spanned records Any VSAM file	Non-VSAM file with variable-length spanned records
UNFORMATTED	Any CMS or MVS file	

IOSTAT=ios

ios is an integer variable or integer array element of length 4. It is set to one of the following values after the I/O statement has been processed:

- Zero, if no error was detected
- Positive, if an error was detected

Refer to Appendix D, "Library Procedures and Messages" on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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. This specifier always returns NO for a striped file. See Figure 27 on page 154 for additional information.

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.

INQUIRE

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.

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=*k/e*

k/e is an integer variable or integer array element of length 4.

If the file is connected for keyed access, the value of *k/e* 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 *k/e* is the position of the leftmost character in the record of the key of the file designated on the INQUIRE statement.

Otherwise, *k/e* 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.

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.

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 up to 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.

NAMED=*nmd*

nmd is a logical variable or logical array element of length 4. This specifier always returns false for a striped file.

If you are using INQUIRE by file name: *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.

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. Note that more than one record can have been read or written as a result of using the slash (/) format code. If the file is connected, but no records have been read or written since the connection was established, *nxr* equals 1.

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.

OPENED=*opn*

opn is a logical variable or logical array element of length 4. *opn* becomes undefined if an error occurs. This specifier allows you to check whether a unit or file is connected, either by the OPEN statement or by preconnection.

If you are using INQUIRE by file name:

When you specify a default ddname in the FILE specifier, the value true will be returned only if the following conditions are met:

- For the form FT*nnFmmm*, the unit number (*nn*) is within the range allowed for your installation and the sequence number (*mmm*) is equal to the number of the particular subfile to which the file is positioned. Subfiles are discussed in more detail in the *VS FORTRAN Version 2 Programming Guide*.
- For the form FT*nnPmmm*, the unit number (*nn*) is within the range allowed for your installation, and the stripe number (*mmm*) is within the range 001 to the total number of stripes.
- For the form FT*nnKkk*, the unit number (*nn*) is within the range allowed for your installation and the key number (*kk*) is within the range of the number of keys for the file that is connected.
- For the form FTERR*sss* or FTPRT*sss*, MTF is active and the subtask number (*sss*) is within the range of the number of active MTF subtasks.

If you are using INQUIRE by unit: When you execute an INQUIRE by unit, the value that will be returned for a unit that is preconnected but for which no I/O statements other than INQUIRE have been executed depends on which of the run-time options, INQPCOPN or NOINQPCOPN, is in effect. When INQPCOPN is in effect, a value of true will be returned if the unit is connected (by an OPEN statement or by preconnection); that is, if an I/O statement other than OPEN, CLOSE, or INQUIRE can be executed without first executing an OPEN statement. When NOINQPCOPN is in effect, a value of false will be returned, even for a preconnected unit, if no I/O statements other than INQUIRE have been executed for that unit.

Note: For INQUIRE by unit on CMS, *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.

PAD=*pad*

pad is a character variable or character array element that is assigned one of these values indicating whether input records will be padded:

YES indicates that short input records will be padded with blanks or that the file is not connected.

NO indicates that short input records will not be padded.

PASSWORD=*pwd*

pwd is a character expression of up to eight characters in length. You must specify the file's read password for *pwd*. For VSAM files that are password-protected, you must code the PASSWORD specifier if you have not already connected the file.

POSITION=*pos*

pos is a character variable or character array element that is assigned one of these values, specifying the current file position for files that are connected for sequential access:

UNDEFINED

indicates that the file is not connected. UNDEFINED is always returned for a file connected for direct or keyed access.

REWIND

indicates that the file is connected and is known to be positioned at the beginning.

APPEND

indicates that the file is connected and is known to be positioned at the end.

ASIS

indicates that the file is connected, and not known to be positioned at either the beginning or the end.

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.

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.

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. This specifier always returns YES if a stripe exists as defined by the EXIST specifier. If the stripe does not exist the SEQUENTIAL specifier returns NO. See Figure 27 on page 154 for additional information.

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.

INQUIRE

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.

concat=noframe=box hdfame=rules.

Figure 29 (Page 1 of 3). 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 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

Figure 29 (Page 2 of 3). 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 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 29 (Page 3 of 3). 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, OR 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
Note: 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.		

Note: There may be system messages issued describing abends. These system messages do not indicate any failure of the program unless a VS FORTRAN message is also issued.

Valid INQUIRE Statements on CMS and MVS:

```
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 99.

INTRINSIC Statement

The INTRINSIC statement identifies a VS FORTRAN-supplied 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 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 can be used with more than one type of argument. Only a specific intrinsic function name can 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, see Chapter 6, “Intrinsic Functions” on page 280.

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.

NAMELIST

The following names of specific intrinsic functions must *not* be passed as actual arguments:

AMAX0	IFIX	MIN1	IQINT
AMAX1	INT	REAL	QCMLPX
AMIN0	LGE	SNGL	QEXT
AMIN1	LGT	CMPLX	QEXTD
CHAR	LLE	DBLE	QFLOAT
DMAX1	LLT	DBLEQ	QMAX1
DMIN1	LOC	DCMLPX	QMIN1
FLOAT	MAX0	DFLOAT	QREAL
ICHAR	MAX1	DREAL	SINGLQ
IDINT	MIN0	HFIX	

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 138.

LOGICAL Type Statement

See “Explicit Type Statement” on page 99.

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

identifies an input/output list and is called 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 can belong to one or more NAMELIST lists.

Dummy variables and dummy array names can appear in a NAMELIST list.

The NAMEDLIST statement must precede any statement function definitions and all executable statements. A NAMEDLIST name must be declared in a NAMEDLIST statement and should be declared only once. The name can appear only in input/output statements.

The NAMEDLIST statement declares a *name* to refer to a particular list of variables or array names. Thereafter, the namelist READ statement and namelist WRITE statement are used to transmit data between the file associated with the unit *un* and the variables specified by the NAMEDLIST name.

The rules for input/output conversion of NAMEDLIST data are the same as the rules for data conversion described in “General Rules for Data Conversion” on page 106. The NAMEDLIST data must be in a special form, described in “NAMELIST Input Data.”

NAMELIST Input Data

To be read using a NAMEDLIST 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 NAMEDLIST name. The NAMEDLIST name must be followed by a blank and must not contain any embedded blanks. This name is followed by data 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 items in an input record is:

- Name = Constant
 - The name can be an array element name or a variable name.
 - The constant can be arithmetic, logical, or character. (If the constants are logical, they can be in the form T or .TRUE. and F or .FALSE.; if the constants are characters, they must be included between apostrophes or quotation marks.)
 - Subscripts must be integer constants.
- Array Name = Set of Constants (separated by commas)
 - The *set of constants* consists of arithmetic, logical, or character constants.
 - 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 NAMEDLIST 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 NAMEDLIST list. The list can contain names of items in COMMON and dummy argument names.

The constants in the input data may be integer, real, complex, logical, or character. Listed below are the rules for coding them:

- Each constant must agree in type with its corresponding item in the NAMELIST statement, as shown below:

**If the data type in
the input list is:**

The constant must be:

Integer	Integer
Real	Real or Integer
Complex	Complex
Character	Character
Logical	Logical
Unsigned	Unsigned integer

Note that for integers and exponents, embedded blanks or trailing blanks between the constant and the following comma, if any, are treated as zeros if BLANK='NULL' is not specified on the OPEN statement.

- Character constants must be entered within apostrophes or quotation marks. To include an apostrophe within apostrophe-delimited character data, use two consecutive apostrophes; similarly for including a quotation mark within quotation-mark-delimited data.
- If the constant is logical, it must be in the form T or .TRUE. and F or .FALSE..
- For an array element, subscripts must be integer constants.
- For an array, the number of constants must be less than or equal to the number of elements in the array.
- You can specify a repetition factor for any constant. For example:
3*2.6

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.

The following example shows the input data for a variable and an array element:

```
R4VAR=12.5E12, I2ARR(2)=-44
```

For an array, code the array name, followed by an equal sign, followed by a series of constants. For example:

```
I2ARR=33, -44
```


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.

NULLIFY Statement

The NULLIFY statement disassociates a pointer variable from an addressed pointee variable and sets the value of the pointer variable to the unassigned state.

Syntax

```
NULLIFY (pointer1 [, pointer2 ...])
```

pointer_name

is the name of the pointer variable to be disassociated from its pointee array or scalar variable.

The NULLIFY statement does not release any storage assigned to the pointee variable.

Example:

```
Pointer (Cpt, C(10,10))
```

```
Nullify (Cpt)
```

Related Statements

- Use the ALLOCATE statement (see “ALLOCATE Statement” on page 54) to assign space for a pointee array.
- Use the DEALLOCATE statement (see “DEALLOCATE Statement” on page 78) to release space for a pointee array.
- Use the POINTER statement (see “POINTER Statement” on page 178) to specify a pointer variable and associate it with a pointee variable.

OPEN Statement

An OPEN statement can 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=rcf]
    [, ACTION=act] [, PASSWORD=pwd]
    [, POSITION=pos]
    [, PAD=pad]
    [, DELIM=dlim]
    [, 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 any VSAM file, an OPEN statement is required. VSAM files cannot be opened in a parallel program under CMS.

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 unless it is for an unnamed file in an

originated task not connected to one of the default units, in which case the status is assumed to be SCRATCH. 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*.
- When dealing with files on READ-ONLY extensions to READ-WRITE disks, STATUS=NEW will cause a file to be created on the parent (or READ-WRITE) disk, even though a file with the same name resides on the extension disk.

FILE=*fn*

fn is a character expression whose value, when any trailing blanks are removed, is used to identify a file. It can take one of the following formats:

- **On CMS and MVS:** *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 CMS:** */filename filetype [filemode] [(member)]*

a character expression whose value, when any trailing blanks are removed, is the filename, filetype, and filemode.

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 where the first character ranges from A to Z and the 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 filemode.
- If an asterisk (*) is specified for the filemode, 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. Use *member* only when you refer to a LOADLIB, MACLIB, or TXTLIB filetype. You cannot 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 can 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 327.

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.
 - FTSExxxx, FTSPxxxx, FTSQxxxx, FTSRxxxx, FTUExxxx, FTUPxxxx, FTUQxxxx, FTURxxxx
- **On MVS:** /dsn [(*member*)]

a character expression whose value, when any trailing blanks are removed, is the data set name on MVS. It can 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 lowercase is equivalent to uppercase.

If this form is used, the FILEINF service routine can 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 327.

- **Other CMS and MVS Considerations:**

If the FILE specifier is omitted, one of the default names for unnamed files is assumed. Default names must not be specified on the FILE specifier. Default names 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 names of this form. In an originated task, unnamed files do not have default ddnames, except for those files connected to the default units, i.e. FTSExxxx, FTUExxxx, etc.
- The ddname for an unnamed file using striped I/O must take the form:

FTnnPmmm

where *nn* is the 2-digit unit number and *mmm* is the 3-digit stripe number. For more information on striped I/O see *VS FORTRAN Version 2 Programming Guide*.

- FTnnKkk, where *nn* is the two-digit unit identifier and *kk* is the two-digit sequence number. Keyed files are assigned default names of this form.
- FTERRsss, where *sss* is the three-digit MTF subtask number. For the error message unit of an MTF subtask, the name defaults to this form.
- FTPRTsss, where *sss* is the three-digit MTF subtask number. For the PRINT/WRITE unit of an MTF subtask, the name defaults to this form. (If the PRINT/WRITE unit and the error message unit are the same, the PRINT/WRITE unit's default name is FTERRsss.)
- FTSExxxx, FTSPxxxx, FTSQxxxx, FTSRxxxx, FTUExxxx, FTUPxxxx, FTUQxxxx, FTURxxxx

Source files that specify system dependent file names in OPEN statement can be compiled on any of the supported operating systems; however, the program can run on only the operating system for which the filename is valid.

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 124 and “BZ Format Code” on page 124, respectively. This specifier is only allowed for formatted I/O.

CHAR=*chr*

chr is a character expression whose value, when any trailing blanks are removed, must be DBCS or NODBCS. 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. A positive value for *ios* signifies that an error has been detected; otherwise, *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

ACTION=*act*

act is a character expression whose value, when any trailing blanks are removed, 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

If the file is to have records written to it and 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 CMS 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 If the 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 nonempty file for retrieval. Update operations cannot be performed on the file.

READWRITE If the file can 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 CMS 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 can execute WRITE statements in addition to READ statements. For keyed access files, you can 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 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=rc/

rc/ 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 with ACCESS='SEQUENTIAL': The following specifiers are used only if ACCESS='SEQUENTIAL'.

POSITION

is a character expression whose value (when any trailing blanks are removed) indicates how the file is to be positioned when OPEN processing is completed.

REWIND

indicates that the file is to be positioned at the beginning, to allow a new file to be written or an existing file overwritten.

APPEND

indicates that the file is to be positioned at the end, to allow records to be added at the end of an existing file.

ASIS

indicates that no specific positioning is to occur. ASIS maintains the position of the file from a prior OPEN or file positioning statement.

PAD

indicates how to process a “short record,” an input record that does not contain enough data for all items in the input list.

YES

indicates that short input records will be padded with blanks to the required length.

NO

indicates that short records will not be padded, and an error condition will be raised.

If PAD is specified on the OPEN statement, the RECPAD run-time option does not apply. If PAD is not specified, the RECPAD run-time option controls the processing of short records.

DELIM

is a character expression whose value, when any trailing blanks are removed, indicates how character constants are delimited in the output file for list-directed or NAMELIST. For compatibility with previous releases of VS FORTRAN, character constants in NAMELIST output are delimited with apostrophes unless the file is opened with the DELIM specifier having the value NONE or QUOTE.

APOSTROPHE indicates that character constants are enclosed in apostrophes. (If the character constant contains an apostrophe, it will be doubled.) This is the default for namelist write.

QUOTE indicates that character constants are enclosed in quotation marks. (If the character constant contains a quotation mark, it will be doubled.)

NONE indicates that character constants are not enclosed in either apostrophes or quotation marks. This is the default for list-directed WRITE.

Specifier Used for VSAM Files**PASSWORD=*pwd***

pwd is a character expression of up to eight characters in length. You must specify the appropriate password for the file. 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. 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.

Note: There may be system messages issued describing abends. These system messages do not indicate any failure of the program unless a VS FORTRAN message is also issued.

Valid OPEN Statements under CMS and MVS:

```
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 Nonconnected 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 can be connected in one program unit of an executable program. Once the unit is connected, it can 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. A failed OPEN, however, will cause the unit to be no longer connected to a file.

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.

PARAMETER

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 can 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=*rc* is not a positive integer.
 - The OPEN statement specifies a different unit from 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.
- OPEN statement for the error message unit.

The OPEN statement for the error message unit can only contain the following specifiers: UNIT, ERR, IOSTAT, and CHAR. Any other specifiers will cause a run-time error.

PARAMETER Statement

The parameter statement assigns a name to a constant.

Syntax

```
PARAMETER ( name1 = constant1 [, name2 = constant2 ... ] )
```

name

is a symbolic name. The name must be defined only once in a PARAMETER statement of a program unit.

constant

is a constant expression.

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.

If the name is of integer, real, double-precision, complex, or unsigned type, the corresponding expression must be a constant arithmetic expression. The exponentiation operator is not permitted unless the exponent is of type integer, unsigned, or byte.

If a name is of character type, the corresponding expression must be a character constant expression.

If a name (*name1*,*name2*) is of logical type, the corresponding expression (*c1*,*c2*) must be a constant logical expression.

Each name is the name of a constant that becomes defined with the value of the expression that appears to the right of the equal sign. The value assigned is determined by the rules used for assignment statements (see Figure 21 on page 58).

Any name of a constant that appears in an expression 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.

The character intrinsic functions, CHAR, ICHAR, and LEN can be used in the constant expression when their arguments are constants. The argument to LEN must not be a dummy argument that has been specified with inherited length.

Valid PARAMETER Statements:

```
CHARACTER*5 C1,C2
PARAMETER (C1='DATE ',C2='TIME ',RATE=2*1.414)

CHARACTER*1 CCNTL
PARAMETER (CCNTL=CHAR(0))

CHARACTER*(*) MSG
INTEGER MSGLEN
PARAMETER (MSG='invalid input')
PARAMETER (MSGLEN = LEN(MSG))

INTEGER IA
PARAMETER (IA=ICHAR('A'))

INTEGER numeric_a
PARAMETER (numeric_a = ICHAR('a'))
```

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.

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

POINTER Statement

The POINTER statement specifies a **pointer variable** and associates it with a target variable, called a **pointee variable**. The value of the pointer variable is the storage address of the pointee variable.

Syntax

POINTER [**len*](*ptr1* [**len*], *ptee1* [(*dim*)])[, (*ptr2* [**len*], *ptee2* [(*dim*))]]...

ptr is the name of the pointer variable.

len

is the length of the pointer, either 4 or 8.

The default length of pointers is 8 but can be reset by the PTRSIZE compiler option. You can use *len* to change that length for individual pointers. See the VS FORTRAN Version 2 Programming Guide for a description of PTRSIZE.

ptee

is the name of the pointee variable or array.

dim

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

(For rules governing dimension bounds, see “Size and Type Declaration of an Array” on page 27.)

A pointer variable is 4 or 8 bytes long and contains the address of another variable, which can be an array or scalar variable.

For computational purposes a pointer variable is of type integer, but you cannot explicitly assign a type to a pointer variable.

The storage is associated with the pointer at execution time by the assignment of an address to the pointer by allocation, the use of the LOC function, or pointer assignment. A reference to a pointee requires that the associated pointer be defined. The pointer can become associated with either static or dynamic storage.

See “Dynamically Dimensioned Arrays” on page 29 for information on dynamically dimensioned arrays.

Uses

Pointer variables **can**:

- Contain addresses in extended common blocks (if and only if the pointer variable is length 8, either through the use of PTRSIZE(8) or *len* being 8)
- Be used in computation wherever integer variables can be used
- Be used in COMMON statements
- Be passed as arguments
- Be specified in a LOCAL statement in a parallel loop or parallel section

Restrictions

Pointer variables **cannot**:

- Be assigned an explicit type
- Be used
 - In DATA statements
 - In a dimension specification
 - In NAMELIST or I/O statements
 - In PARAMETER statements
 - In EXTERNAL statements
 - In INTRINSIC statements
 - In explicit type statements
 - In statement functions
 - In SAVE statements
 - As ENTRY, FUNCTION, or PROGRAM names.
- Be dimensioned
- Be pointee variables. In other words, you cannot use one pointer variable to address another pointer variable.

Example:

```
Pointer (Cpt, C(10,10)), (Dpt, D(12)), (Dpt, DD(I1, K1))
Real*4 Buffer(100)
Cpt = LOC(Buffer)
Read(5, '(10F10.2/)' ) C
```

Related Statements

- Use the ALLOCATE statement (see “ALLOCATE Statement” on page 54) to assign space for a pointee array.
- Use the DEALLOCATE statement (see “DEALLOCATE Statement” on page 78) to release space for a pointee array.
- Use the NULLIFY statement (see “NULLIFY Statement” on page 167) to disassociate a pointer variable from a pointee variable.

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 181
3. “PRINT Statement—NAMELIST with External Devices” on page 182

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, and its function is more 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 the default unit as set for the installation or by the RDRUNIT run-time option.

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 186.

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 omitted a blank record is written. See “Implied DO in an Input/Output Statement” on page 90.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

Data Transmission: In the absence of the \$ format control, 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 each item in the list in the order they are specified. The number of character data specified by the format code is taken from the list, converted according to the corresponding format code, and 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, B, O, or Z, or is empty (that is, FORMAT()), a blank record is written out, in the absence of \$ format control.

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 243.

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 the default unit as set for the installation or by the RDRUNIT run-time option.

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 90.

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed. If *list* is omitted, a blank record is written.

For more information, see “WRITE Statement—List-Directed I/O to External Devices” on page 249.

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 164.

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 254.

PROGRAM Statement

The PROGRAM statement assigns a name to a main program. It must be the first statement in the main program.

If the program is a parallel program, then the PROGRAM statement defines the beginning of the root task.

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 can appear; it has the same effect as a STOP statement.

The PROGRAM statement can be used only 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 the compiler to be MAIN#. No recompilation is necessary for existing programs with the main program name MAIN. Any linkage editor control statement referencing the name MAIN for a main program that is recompiled with VS FORTRAN Version 2 Release 5 must be changed to 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 186
3. "READ Statement—Formatted with Keyed Access" on page 189
4. "READ Statement—Formatted with Sequential Access" on page 193
5. "READ Statement—Formatted with Sequential Access to Internal Files" on page 196
6. "READ Statement—List-Directed I/O from External Devices" on page 200
7. "READ Statement—List-Directed I/O with Internal Files" on page 202
8. "READ Statement—NAMELIST with External Devices" on page 204
9. "READ Statement—NAMELIST with Internal Files" on page 206
10. "READ Statement—Unformatted with Direct Access" on page 209
11. "READ Statement—Unformatted with Keyed Access" on page 211
12. "READ Statement—Unformatted with Sequential Access" on page 215

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 can 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 with a value equal to or greater than zero.

un is a required identifier that optionally can be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. If UNIT=

READ (Asynchronous)

is included on the READ statement, all the specifiers can appear in any order. The other 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 can 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 can 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 can 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* cannot 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.

Arrays in extended common blocks cannot appear in *list*.

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 cannot 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* bytes of the record are 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 can receive fewer than *len* bytes. If the record length is greater than the receiving area size, an error is detected.

The specified array can 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 can be executed between an asynchronous READ and its corresponding WAIT, subject to the following rules:

- No array element in the receiving area can 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 can appear that redefines or undefines a variable or array element appearing in the subscript of *e1* or *e2*. See “Valid and Invalid Programs” on page 2.
- If a function reference appears in the subscript expression of *e1* or *e2*, the function cannot be referred to by any statements executed between the asynchronous READ and the corresponding WAIT. Also, no subroutines or functions can 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 can 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 are 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 168).

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 with a value greater than or equal to zero. *un* is a required identifier that optionally can 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 READ statement FMT= must be used, and all the specifiers can appear in any order.

FMT=*fmt*

fmt is a required format identifier.

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

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 or quotation marks, 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. A quotation mark in a constant enclosed in quotation marks is represented by two consecutive quotation marks.

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 can precede the left parenthesis and character data can 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 can 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* can 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* can 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 32.)

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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

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*.

READ (Formatted, Direct Access)

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 to 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, B, O, 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 treated as transmission errors unless the CNVIOERR run-time option is specified.

If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines can be used to detect and handle these errors. See Chapter 11, “Extended Error-Handling Topics” on page 382.

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 168.)

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=kgf]
    [, 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 identifier.

un is an integer expression of length 4 with a value greater than or equal to zero. It is a required identifier that optionally can 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 included, and all the specifiers except list can appear in any order.

FMT=fmt

fmt is a required format identifier.

fmt optionally can be preceded by FMT=. If FMT is omitted, the format identi-

READ (Formatted, Keyed Access)

fier 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 186.

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. A positive value for *ios* signifies that an error has been detected. A negative value for *ios* signifies that end of file has been detected. A zero value for *ios* signifies that no error has been detected. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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=<i>key</i>	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=<i>kge</i>	Specifies the following search criterion for the record to be retrieved: If the file contains a record whose key value is identical to <i>kge</i> , the first such record is retrieved. If not, the first record with the next greater key value is retrieved. If <i>kge</i> 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=<i>kgt</i>	Specifies that the record to be retrieved is the first one with a key value greater than <i>kgt</i> . If <i>kgt</i> 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 90.

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 trans-

mission 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, B, O, 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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines can be used to detect and handle these errors. (See Chapter 11, "Extended Error-Handling Topics" on page 382.)

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 can 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 49.)

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 the default unit as set for the installation or by the RDRUNIT run-time option.

un is a required identifier in the first form of the READ statement. It optionally can 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 READ statement, FMT= must be used and all the specifiers can appear in any order.

READ (Formatted, Sequential Access)

In the first form of the READ in which *un* is not specified, the unit is the default unit as set for the installation or by the RDRUNIT run-time option.

FMT=*fmt*

fmt is a required format identifier.

It optionally can 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 186.

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. A positive value for *ios* signifies that an error has been detected. A negative value for *ios* signifies that end of file has been detected. A zero value for *ios* signifies that no error has been detected. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.) 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 a comma or end of the record lies within the range of the format field when dealing with non-character input, the field width is reduced to the size up to, but not including, the comma and up to the end of the record. The next field begins immediately after the comma or after the end of the record. If additional data is required from the record when the current field pointer is outside the record, then an error message is issued.

If the record contains more data than is necessary to satisfy all the subsequent 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 subsequent items of the list and the associated format identifier, an error message will be issued. If there is only one item in the list and the record does not contain enough data to satisfy that item, the data will be considered FALSE for format code L, and 0 for codes I, D, E, F, G, Q, B, O, and Z.

If the list is not specified and the format identifier starts with an I, E, F, D, G, L, Q, B, O, 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.

READ (Formatted, Sequential Access, Internal)

Errors caused by the length of the data record or the value of the data are not treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 11, “Extended Error-Handling Topics” on page 382.)

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

un is a required identifier that optionally can 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 except *list* can appear in any order.

FMT=*fmt*

fmt is a required format identifier.

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

If FMT= is omitted, *fmt* must appear second. If both UNIT= and FMT= are included in the READ statement, all the specifiers except *list* can appear in any order.

For explanations of these format identifiers, see “READ Statement—Formatted with Direct Access” on page 186.

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. A positive value for *ios* signifies that an error has been detected. (Refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value.) A negative value for *ios* signifies that end of file has been detected. A zero value for *ios* signifies that no error has been detected. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

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.

READ (Formatted, Sequential Access, Internal)

Valid READ Statements:

```
READ (CHARVR, 100) ((A(I,J),J=1,20),B(I),I=1,10)
```

```
READ (CHARVR, FMT=100) ((A(I,J),J=1,20),B(I),I=1,10)
```

```
READ (UNIT=CHARVR, FMT=100) ((A(I,J),J=1,20),B(I),I=1,10)
```

Invalid READ Statements:

```
READ (FMT=100, CHARVR) ((A(I,J),J=1,20),B(I),I=1,10)  un must appear first because  
UNIT= is not specified.
```

```
READ (100, UNIT=CHARVR) ((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, the record is logically extended with zeros or blanks in order to satisfy the remaining items.

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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to

determine the number of the error that was detected. The extended error handling subroutines can be used to detect and handle these errors. (See Chapter 11, “Extended Error-Handling Topics” on page 382.)

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.

Valid Internal File Example:

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

```

READ Statement—List-Directed I/O from External Devices

This statement transfers data from an external device into internal storage. The specifiers used 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 49.)

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 the default unit as set for the installation or by the RDRUNIT run-time option.

un is a required identifier in the first form of the READ statement. It optionally can be preceded by UNIT=. If UNIT= is omitted, *un* must appear immediately following the left parenthesis. If UNIT= is included in the READ statement, all the specifiers can appear in any order.

In the form of the READ in which *un* is not specified, the unit is the default unit as set for the installation or by the RDRUNIT run-time option.

FMT= *

specifies that a list-directed READ is to be executed.

It optionally can 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 external file is encountered, control is transferred to *stl*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. A positive value for *ios* signifies that an error has been detected. A negative value for *ios* signifies that end of file has been detected. A zero value for *ios* signifies that no error has been detected. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors

set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

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)	<u>un</u> 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 first 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 treated as transmission errors unless the CNVIOERR run-time option is specified.

READ (List-Directed, Internal)

If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines can be used to detect and handle these errors. (See Chapter 11, “Extended Error-Handling Topics” on page 382.)

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

un is a required reference that optionally can 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 optionally can be preceded by FMT=. If FMT= is omitted, * must appear second. If both UNIT= and FMT= are included in 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. A positive value for *ios* signifies that an error has been detected. (Refer to Appendix D, "Library Procedures and Messages" on page 438 for messages that correspond to the IOSTAT value.) A negative value for *ios* signifies that end of file has been detected. A zero value for *ios* signifies that no error has been detected. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

Valid READ Statements:

```
READ (CHARVR,*) ACTUAL(1)
READ (CHARVR,FMT=*) ACTUAL(1)
READ (FMT=*,UNIT=CHARVR) 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 168).

Syntax

READ *name*

READ

```
( [UNIT=]un,
  { [FMT=]name | [NML=]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 the default unit as set for the installation or by the RDRUNIT run-time option.

un is required in the second form of the READ statement. It optionally can 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 READ statement, FMT= or NML= 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 the default input unit as set for the installation or by the run-time option RDRUNIT.

FMT=name

NML=name

name is a NAMELIST name. See "NAMELIST Statement" on page 164.

If the keyword is omitted, the NAMELIST name must appear second, and UNIT= must not be used.

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. A positive value for *ios* signifies that an error has been detected. A negative value for *ios* signifies that end of file has been detected. A zero value for *ios* signifies that no error has been detected. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, "Library Procedures and Messages" on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 165.

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 63 and “REWIND Statement” on page 219).

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 165.)

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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines can be used to detect and handle these errors. (See Chapter 11, “Extended Error-Handling Topics” on page 382.)

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 | [NML=]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

un is a required reference. It optionally can 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= or NML= must be used and all the specifiers can appear in any order.

FMT=*name***NML=*name***

name is a NAMELIST name. See “NAMELIST Statement” on page 164.

If the keyword is omitted, the NAMELIST name must appear second, and UNIT= must not be used.

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. A positive value for *ios* signifies that an error has been detected. (Refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value.) A negative value for *ios* signifies that end of file has been detected. A zero value for *ios* signifies that no error has been detected. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 165.

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 165.)

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 168.) 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. *un* is a required identifier that optionally can 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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 11, “Extended Error-Handling Topics” on page 382.)

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 168.)

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.

un is a required identifier that optionally can 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. A positive value for *ios* signifies that an error has been detected. A negative value for *ios* signifies that end of file has been detected. A zero value for *ios* signifies that no error has been detected. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors

set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 11, "Extended Error-Handling Topics" on page 382.)

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 49.)

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 a required identifier that optionally can 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

READ (Unformatted, Sequential Access)

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. A positive value for *ios* signifies that an error has been detected. A negative value for *ios* signifies that end of file has been detected. A zero value for *ios* signifies that no error has been detected. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 11, “Extended Error-Handling Topics” on page 382.)

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 99.

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.

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 224), in a scheduled subroutine become undefined when the task environment is terminated. Those in a subprogram activated by a PARALLEL CALL statement, or within a parallel loop or section, become undefined at the end of the thread.
- 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, except a block local to a parallel thread (that is, specified in the subprogram but not in the referencing program). Entities in a blank common block of a scheduled subroutine become undefined when the task terminates.
- 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. The entities in a named common block may become undefined by execution of a RETURN or END statement in another program unit. Entities in a named common block local to a parallel thread become undefined at the end of the thread. Entities in a named common block of a scheduled subroutine become undefined when the task terminates.

When a subprogram is referenced in a non-parallel environment, the RETURN statement associated with that subprogram must be executed before any other reference to that subprogram name.

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 131.)

Syntax**RETURN****RETURN Statement in a Subroutine**

A subroutine 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 229.)

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 $1 \leq m \leq n$, 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.

Note: When a subroutine is referenced by a PARALLEL CALL or SCHEDULE statement, *m* is ignored and control is returned to the calling routine when a corresponding WAIT FOR statement is executed.

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 168.)

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*

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*. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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*.

REWRITE (Formatted, Keyed Access)

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 “READ Statement—Formatted with Direct Access” on page 186.

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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

REWRITE (Unformatted, Keyed Access)

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 90.) 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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines can be used to detect and handle these errors. (See Chapter 11, “Extended Error-Handling Topics” on page 382.)

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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.) 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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines can be used to detect and handle these errors. (See Chapter 11, "Extended Error-Handling Topics" on page 382.)

SAVE Statement

The SAVE statement retains the definition status of an entity (a named common block, variable, or array) after the execution of a RETURN or END statement in a subprogram.

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, names of entities in a common block and pointee names must not appear in a SAVE statement.

A SAVE statement without a list is treated as though it contained the names of all allowable 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.

Under CMS and MVS, all entities are saved. In parallel programs, users should be aware that there may be multiple copies of saved variables due to multiple parallel tasks and privatization of variables for concurrent processing. A subroutine or function invoked within a parallel loop or section or a subroutine invoked with a PARALLEL CALL or SCHEDULE statement will have local variables that are private to the parallel thread.

The values of saved variables in a subroutine invoked with the SCHEDULE statement are retained across multiple schedules to the same task. (The values of variables initialized with the DATA statement are available only to the first invocation of the SCHEDULEd task. Subsequent invocations will see values as modified by earlier executions.)

The values of saved variables in a subroutine invoked with the PARALLEL CALL statement or a subroutine or function invoked within a parallel loop or section are retained across multiple invocations within the same thread (for example, the same iteration of a loop). (The values of variables initialized with the DATA statement are available only to the first invocation of the subprogram invoked by each thread within a parallel program.) The particular set of values for saved variables used when new parallel constructs are processed is indeterminate and may vary among successive runs.

Within a function or subroutine, an entity with the saved attribute does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. This entity is defined with the same value at the next reference of that subprogram. See "RETURN Statement" on page 217 for exceptions in parallel processing.

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

$$\textit{name} \left([\textit{arg1} [, \textit{arg2} \dots]] \right) = \textit{m}$$

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, 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 32.)

Statement Function

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 can 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 its 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:

$C = \text{FUNC}(D, E)$

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
$\text{SUM}(A, B, C, D) = A + B + C + D$	$\text{NET} = \text{GROS-SUM}(\text{TAX}, \text{COVER}, \text{HOSP}, \text{STOC})$ $\text{BIGSUM} = \text{SUM}(A, B, \text{SUM}(C, D, E, F), G(I))$
$\text{FUNC}(Z) = A + X * Y * Z$	$\text{ANS} = \text{FUNC}(\text{RESULT})$
$\text{VALID}(A, B) = .\text{NOT. } A .\text{OR. } B$	$\text{VAL} = \text{TEST} .\text{OR. } \text{VALID}(D, E)$
$\text{VOLUME}(R) = 4.0 * \text{PI} / 3.0 * R ** 3$	$\text{TOTVOL} = \text{VOLUME}(R1) + \text{VOLUME}(R2)$

Invalid Statement Function Definitions:

$\text{SUBPRG}(3, J, K) = 3 * I + J ** 3$	Arguments must be variables.
$\text{SOMEF}(A(I), B) = A(I) / B + 3.$	Arguments must not be array elements.
$3\text{FUNC}(D) = 3.14 * E$	Function name must begin with an alphabetic character.
$\text{BAD}(A, B) = A + B + \text{BAD}(C, D)$	A recursive definition is not permitted.
$\text{NOGOOD}(A, A) = A * A$	Arguments are not distinct variable names.
$\text{IFLUNK}(I) = \text{IDBLI}(I)$	Function IDBLI changes the value of argument I.

Invalid Statement Function References:

(The functions are defined as above.)

$\text{WRONG} = \text{SUM}(\text{TAX}, \text{COVER})$	Number of arguments does not agree with above definition.
$\text{MIX} = \text{FUNC}(I, J)$	Types of the arguments do not agree with above definition.
$\text{MYGRAD}(I) = \text{IFLUNK}(I) + I$	I is modified by function IFLUNK(I).

Statement Labels

Statement labels identify statements in your source programs. Any statement can have a label in either fixed form or free form. See “Source Language Statements” on page 9.

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 55.

STATIC Statement

The STATIC statement identifies the variables and arrays to be assigned the static storage class.

Syntax

```
STATIC name1 [/clist1] [, name2[/clist2] ... ]
```

name

is a variable or array. It has the form:

```
a[(dim)]
```

where:

a is a variable or array

dim

is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array. *dim* can be specified only for arrays.

(For rules governing dimension bounds, see “Size and Type Declaration of an Array” on page 27.)

clist

is a list of constants or the names of constants to provide an initial value for the variable or array specified by the immediately preceding name. Initialization occurs as described in “DATA Statement” on page 76.

One type of storage class is static. For each data object that is declared static, there is exactly one copy for all invocations of the defining program unit.

The same name must not appear in more than one AUTOMATIC or STATIC statement. Dummy argument names, procedure names, names of constants, and

names of pointees must not appear in STATIC statements. Names declared as STATIC also cannot appear in COMMON (as an item), SAVE, EXTERNAL, or INTRINSIC statements. Any appearance of *name* in an explicit type statement must precede the STATIC statement.

Local variables and arrays are static by default.

STOP Statement

The STOP statement ends the processing of the object program and may display a message.

Syntax

STOP [*n*]

n is a string of 1 through 5 decimal digits, or a character constant of up to 72 characters in length.

If you are running under MVS, the value you supply to the STOP statement is returned as the condition code for the job step being processed. If you code a value greater than 4095, the condition code returned may not match the value you specified in the STOP subroutine.

If you are running under CMS or MVS in a REXX EXEC and you use a decimal value for the STOP statement, the value is returned to your EXEC as the contents of variable RC.

When the program processes the STOP statement, operator message AFB002I is displayed at the console.

A STOP statement cannot terminate the range of a DO-loop.

Valid STOP Statements:

```
STOP
STOP 4095
STOP 'PROGRAM BACGAM EXECUTION COMPLETED'
```

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 can 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, 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. 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 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 27.

The subroutine 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, see “END Statement” on page 91 and “RETURN Statement” on page 217.

Actual Arguments in a Subroutine

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 (not if within parallel loop or section)
- An external procedure name (not if within parallel loop or section)

- 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

The dummy arguments are replaced at the time of processing of the CALL, PARALLEL CALL, or SCHEDULE statement by the actual arguments supplied in the calling statement.

Dummy arguments must follow certain rules:

- None of the dummy argument names can appear in an EQUIVALENCE, COMMON (except as a common block name), DATA, PARAMETER, SAVE, INTRINSIC, AUTOMATIC, STATIC, or POINTER (except as a pointer name) statement.
- 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 is part of static debug, which is not recommended for use. Unless you are working with an older version of VS FORTRAN, you should use interactive debug. For more information, see the *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

WAIT

Note: The TRACE OFF statement cannot be used with extended common. For more information on extended common, see the *VS FORTRAN Version 2 Programming Guide*.

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 is a part of the VS FORTRAN static debug that is not recommended for use. Unless you are working with an older version of Fortran, you should use the VS FORTRAN interactive debug to debug your programs. For more information on VS FORTRAN interactive debug, see the *VS FORTRAN Version 2 Interactive Debug Guide and Reference*.

Note: The TRACE ON statement cannot be used with extended common. For more information on extended common, see the *VS FORTRAN Version 2 Programming Guide*.

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 79.) 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 135.

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 *e1* or *e2*, 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.

Arrays in extended common blocks cannot appear in *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:

“WRITE Statement—Asynchronous” on page 237.

“WRITE Statement—Formatted with Direct Access” on page 238.

“WRITE Statement—Formatted with Keyed Access” on page 241.

“WRITE Statement—Formatted with Sequential Access” on page 243.

“WRITE Statement—Formatted with Sequential Access to Internal Files” on page 246.

“WRITE Statement—List-Directed I/O to External Devices” on page 249.

“WRITE Statement—List-Directed I/O with Internal Files” on page 252.

“WRITE Statement—NAMELIST with External Devices” on page 254.

“WRITE Statement—NAMELIST with Internal Files” on page 256.

“WRITE Statement—Unformatted with Direct Access” on page 257.

“WRITE Statement—Unformatted with Keyed Access” on page 260.

“WRITE Statement—Unformatted with Sequential Access” on page 262.

WRITE Statement—Asynchronous

The asynchronous WRITE statement transmits unformatted 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 with a value greater than or equal to zero. It is a required identifier that optionally can 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.

Arrays in extended common blocks cannot appear in *list*.

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*.

WRITE (Formatted, Direct Access)

- 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 *e1* or *e2*.
- If a function reference appears in the subscript expression of *e1* or *e2*, 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 49.)

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 with a value greater than or equal to zero. It is a required identifier that optionally can 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.

If FMT= is omitted, the format identifier *fmt* 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 186.

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*.

WRITE (Formatted, Direct Access)

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90. If *list* is omitted, a blank record is written.

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, B, O, 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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines can be used to detect and handle these errors. (See Chapter 11, “Extended Error-Handling Topics” on page 382.)

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=st]
    [, IOSTAT=ios]
    [, DUPKEY=st]])
  list
```

UNIT=*un*

un is the external unit identifier.

un is an integer expression of length 4 with a value greater than or equal to zero. It is a required identifier that optionally can 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 that optionally can 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

WRITE (Formatted, Keyed Access)

- A character expression
- An array name

For explanations of these format identifiers, see “READ Statement—Formatted with Direct Access” on page 186.

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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 11, “Extended Error-Handling Topics” on page 382.)

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 49.)

Syntax

```
WRITE
  ( [UNIT=]un,
    [FMT=]fmt
    [, ERR=stf]
    [, IOSTAT=ios] )
    [lstf]
```

WRITE (Formatted, Sequential Access)

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 the default unit as set for the installation or by the PRTUNIT run-time option.

un is a required identifier that optionally can 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 that optionally can 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 186.

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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

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, B, O, 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 104.)

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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 11, "Extended Error-Handling Topics" on page 382.)

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 a required reference that optionally can 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.

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*, can 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 238.

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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. Refer to Appendix D, “Library Procedures and Messages” on page 438

for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

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 (CHARVR,100) (A(I),I=1,5),B
```

```
WRITE (CHARVR,FMT=100) (A(I),I=1,5),B
```

```
WRITE (FMT=100,UNIT=CHARVR) (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,CHARVR) (A(I),I=1,5),B
```

un must appear first because UNIT= is not specified.

```
WRITE (100,UNIT=CHARVR) (A(I),I=1,5),B
```

fmt must appear second because FMT= is not 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, B, O, 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.

WRITE (Formatted, Sequential Access, Internal)

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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines may be used to detect and handle these errors. (See Chapter 11, "Extended Error-Handling Topics" on page 382.)

Valid Internal File Example:

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 168.) 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 with a value greater than or equal to zero, or
- An asterisk (*) representing the default unit as set for the installation or by the PRTUNIT run-time option.

un is a required identifier that optionally can 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 optionally can 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*, 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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90.

WRITE (List-Directed, External)

A function must not be referenced within an expression if such a reference causes an input or output statement to be executed.

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 pre-connected 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.

PRINT: The remainder of this section applies to list-directed PRINT as well as list-directed WRITE.

The type and length of the data determine how they will be formatted (see Figure 30). Real numbers are written in G format (see Figure 23 on page 104). For noncharacter data, the constants are each followed by one space.

Figure 30. List-Directed Output Formats

If the data type and length are:	The field width will be:	The fraction will be:	The scale factor will be:	The exponent (if printed) will be:
Real, length of 4	16 bytes	9 digits	0	2 digits
Real, length of 8	25 bytes	18 digits	0	2 digits
Real, length of 16	42 bytes	35 digits	0	2 digits
Logical, length of 1, 2, 4, or 8	1 character	N/A	N/A	N/A
Integer, length of 1	4 bytes	N/A	N/A	N/A
Integer, length of 2	6 bytes	N/A	N/A	N/A
Integer, length of 4	11 bytes	N/A	N/A	N/A
Integer, length of 8	20 bytes	N/A	N/A	N/A
Unsigned, length of 1	3 bytes	N/A	N/A	N/A
Byte	4 bytes	N/A	N/A	N/A
Complex, length of 8	35 bytes	9 digits	0	2 digits
Complex, length of 16	53 bytes	18 digits	0	2 digits
Complex, length of 32	87 bytes	35 digits	0	2 digits
Character, length of <i>n</i>	<i>n</i> bytes	N/A	N/A	N/A

Note:

1. The minimum number of digits written for integers will be 1.

If you code the following PRINT statement

```
PRINT *, R4VAR, I2ARR, CH4
```

and you have declared the data types for the variables and arrays as follows:

```
REAL*4      R4VAR      / -12.5E+12 /
INTEGER*2    I2ARR(2)   / 33, -44 /
CHARACTER*4  CH4        / 'TWO' /
```

the following record will be written:

-0.125000000E+14	33	-44 TWO
------------------	----	---------

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.

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.

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 can 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 or quotation marks by default. The DELIM= specifier of the OPEN statement can be used to override this default.
- Are not preceded or followed by any separators (including blanks)
- Have each internal apostrophe or quotation mark represented externally by two apostrophes or quotation marks only if the DELIM= specifier in effect is "apos-trophe" or "quote", respectively
- 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 treated as transmission errors unless the CNVIOERR run-time option is specified. If CNVIOERR is specified and there is no user error corrective routine, you can code the ERR specifier to pass control to another statement or specify IOSTAT to determine the number of the error that was detected. The extended error handling subroutines may be used to detect and handle these errors. See Chapter 11, "Extended Error-Handling Topics" on page 382.

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] )  
    [/istl]
```

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

un is a required reference that optionally can 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 optionally can 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. A positive value for *ios* signifies that an error has been detected. A zero value for *ios* signifies that no error has been detected. Refer to Appendix D, "Library Procedures and Messages" on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90. If LIST is omitted, a blank record is written.

Valid WRITE Statements:

```
WRITE (CHARVR, *) FIFTY5, ISEG
WRITE (CHARVR, FMT=*) FIFTY5, ISEG
WRITE (FMT=*, UNIT=CHARVR) 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 is 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.

See the *VS FORTRAN Version 2 Programming Guide* for information on how to calculate the size of a record needed to hold all the converted list items. The size of the record will be the sum of the field widths plus a byte to separate each field.

WRITE (NAMELIST, External)

```
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 | [NML=]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 the default unit as set for the installation or by the PRTUNIT run-time option.

un 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= or NML= must be used; and all the specifiers can appear in any order.

FMT=*name*

NML=*name*

name is a NAMELIST name.

See “NAMELIST Statement” on page 164.

If the keyword is omitted, the NAMELIST name must appear second, and UNIT= must not be used.

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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do

they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 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 63).

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 the external file.

See the *VS FORTRAN Version 2 Programming Guide* for information on how to calculate the size of the record on the external file.

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 11, “Extended Error-Handling Topics” on page 382.)

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 | [NML=name
    [, ERR=stl]
    [, IOSTAT=ios] )
```

UNIT=*un*

un is the reference to an area of internal storage called an internal file.

un must be the name of a character array with at least three elements. It is a required reference that optionally can 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= or NML= must be used and all the specifiers can appear in any order.

FMT=*name***NML=*name***

name is a NAMELIST name.

See “NAMELIST Statement” on page 164. If the keyword is omitted, the NAMELIST name must appear second, and UNIT= must not be used.

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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. Refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

Valid WRITE Statements:

```
WRITE (SEVSEV,SUNSET_STRIP)
```

```
WRITE (ARRAY, 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.

See the *VS FORTRAN Version 2 Programming Guide* for information on how to calculate the size of the internal file. 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 168.)

Syntax

```
WRITE
  ( [UNIT=un,
    REC=rec
    [, ERR=st]
    [, IOSTAT=ios]
    [, NUM=n] )
  [list]
```

UNIT=*un*

un is the external unit identifier. *un* is an integer expression of length 4 with a value greater or equal to zero.

un is a required identifier that optionally can be preceded by UNIT=. If UNIT= is included on the WRITE statement, all the specifiers can appear in any order.

If UNIT= is omitted, *un* must appear immediately following the left parenthesis. The other specifiers may 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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Refer to Appendix D, "Library Procedures and Messages" on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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 90. 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 11, "Extended Error-Handling Topics" on page 382.)

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 168).

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 with a value greater than or equal to zero. It is a required identifier that optionally can 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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the IOSTAT=*ios* specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the IOSTAT value. (Not all VS FORTRAN Version 2 errors set the IOSTAT field, nor do they all branch to the ERR label. For a complete list, see Figure 59 on page 390.)

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” on page 261.

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 90.

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 11, “Extended Error-Handling Topics” on page 382.)

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

WRITE (Unformatted, Sequential Access)

- 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 49).

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 with a value greater than or equal to zero. It is a required identifier that optionally can 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. A positive value for *ios* signifies that an error has been detected, otherwise *ios* is set to zero. If you are using VSAM, use the `IOSTAT=ios` specifier to see VSAM return and reason codes. Otherwise, refer to Appendix D, “Library Procedures and Messages” on page 438 for messages that correspond to the `IOSTAT` value. (Not all VS FORTRAN Version 2 errors set the `IOSTAT` field, nor do they all branch to the `ERR` label. For a complete list, see Figure 59 on page 390.)

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 90.

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 11, “Extended Error-Handling Topics” on page 382.)

Chapter 5. Parallel Language Constructs

The following chapter describes language constructs which can be used to write explicit parallel programs. The language constructs described are used to identify where your program is eligible for execution by multiple processors concurrently, or in parallel. The parallel statements are provided for users that are interested in reducing the elapsed execution time of their Fortran programs.

There are four parallel language constructs: parallel task management statements, parallel loop statements, parallel sections statements, and parallel call statements.

Subroutines specified on the PARALLEL CALL statement and subroutines and functions referenced within a parallel loop or parallel section will share units and files with the parallel task in which they execute. Refer to the “SAVE Statement” on page 224 for a description of subprogram local variables and arrays being private.

You can use library lock services to control access to arguments and common items to ensure valid concurrent processing of parallel threads.

Refer to the *VS FORTRAN Version 2 Programming Guide* for an overview of parallel processing and a definition of terms used to define the parallel language constructs.

Parallel Task Management Statements

The parallel task management statements are used to create, schedule, coordinate, and terminate parallel tasks in your programs.

The parallel task management statements are:

- ORIGINATE TASK | ORIGINATE ANY TASK
- SCHEDULE TASK | SCHEDULE ANY TASK
- WAIT FOR TASK | WAIT FOR ANY TASK | WAIT FOR ALL TASKS
- TERMINATE TASK

ORIGINATE Statements

An ORIGINATE statement is used to create a parallel task.

Syntax

ORIGINATE TASK *ptaskid* | **ORIGINATE ANY TASK** *rtaskid*

ptaskid

is an integer variable or an array element of length 4, or an integer constant, whose value is a positive integer. The value you provide for *ptaskid* is the identifier for an originated task. This identifier must be unique within your application, and must represent a value between 1 and 9999, inclusive.

rtaskid

is an integer variable or an array element of length 4. The value returned to *rtaskid* is the identifier for the originated task, and can change in value from one run of a parallel program to another. The value returned will be outside the range of 1 to 9999.

Ptaskid and *rtaskid* are referred to as the originated task identifiers. The task identifier is used by SCHEDULE, WAIT FOR ANY TASK, WAIT FOR TASK, and TERMINATE statements. It is also used in determining the default file names and units associated with the task.

The originated task identifier is used for identification purposes only; its value should not be used in computations.

Multiple levels of parallel tasks can be created. The ORIGINATE statement can be used at any level within a parallel program. A task created by the ORIGINATE statement is *owned* by the task that executed the ORIGINATE statement. When multiple levels of tasks are created, a hierarchical task ownership structure is established. The root task is where the main program begins execution. It is the highest level in the task structure.

Example 1:

```
ORIGINATE ANY TASK ITASK
```

Example 2:

```
INTEGER NTASK(10)
DO I = 1,10
    ORIGINATE ANY TASK NTASK(I)
ENDDO
```

Example 3:

```
ORIGINATE TASK 2
```

SCHEDULE Statements

The SCHEDULE statement assigns a subroutine for asynchronous execution in the environment of a previously originated task. The code contained in the subroutine is a parallel thread which will execute on one or more virtual processors. The scheduling routine continues to execute within its own parallel task.

There are two forms of the SCHEDULE statement:

- The SCHEDULE TASK statement specifies the identifier of the currently unscheduled originated task that is to execute the subroutine.
- The SCHEDULE ANY TASK statement returns the identifier of the originated task selected in which the subroutine will execute. The originated task selected is one that is not currently scheduled.

Every SCHEDULE statement must have a corresponding WAIT FOR statement. An error is detected at execution time if it does not. The SCHEDULE statement can be used if there is a subroutine that can be executed concurrently with the executing program to be done, when an originated task is not currently scheduled.

The originated task being scheduled must be owned by the same parallel task under which the SCHEDULE statement is issued. An error is detected at execution time if the originated task is not owned by the same parallel task, or if the originated task does not exist.

Syntax

```
{ SCHEDULE TASK ptaskid | SCHEDULE ANY TASK rtaskid }  
  [,SHARING (shrcom [,shrcom]...)]  
  [,COPYING (cpcom [,cpcom]...)]  
  [,COPYINGI (cpicom [,cpicom]...)]  
  [,COPYINGO (cpocom [,cpocom]...)]  
  ,CALLING subx [[arg [,arg]...]]
```

ptaskid

is an integer variable or an array element of length 4, or an integer constant. The value you provide for *ptaskid* is the identifier for the originated task that is to process the subroutine, *subx*.

rtaskid

is an integer variable or an array element of length 4. The value returned for *rtaskid* is the identifier of a currently unscheduled originated task that is to process the subroutine, *subx*.

shrcom

is the name of a common block in the scheduling task to be shared with the scheduled task. Sharing is no longer in effect after the scheduled routine completes execution.

cpcom

is the name of a common block in the scheduling task. The common block is copied to the common block of the same name in the scheduled task before the scheduled routine begins to run. After the scheduled routine finishes running, the contents of the common block are copied back into the scheduling task's common block. The second copy operation is performed as part of the processing of the WAIT FOR statement that detects the completion of the scheduled task's processing.

cpicom

is the name of a common block in the scheduling task. The common block is copied to the common block of the same name in the scheduled task before the scheduled routine begins to run. The contents of the common block are *not* copied back into the scheduling task's common block.

cpocom

is the name of a common block in the scheduling task. The common block is copied from the common block of the same name in the scheduled task after the scheduled routine completes execution. The copy operation is performed as part of the processing of the WAIT FOR statement that detects the completion of the scheduled task's processing.

subx

is the name of the subroutine to be processed in parallel.

arg

is an argument that is supplied to the subroutine. The argument can be a variable, array element, array name, constant, or expression (including a parenthesized primary). The argument cannot be a function name, a subroutine name, or an asterisk (*) followed by a statement label.

Note: See “Common Blocks in Parallel Programs” on page 74 within the “COMMON Statement” on page 73 section for more information about common blocks and parallel.

Arguments that are expressions or current DO variables are passed by copy. Other data passed as arguments are shared. In order to preserve computational independence, the same variables or array elements should not be referenced or modified by the scheduling routine until a WAIT FOR statement is specified, unless appropriate control mechanisms are used.

Example 1:

```
SCHEDULE ANY TASK IT3, SHARING (MYCOM),
+ CALLING SUBC(D,E,F), COPYING (COM2)
```

Example 2:

```
SCHEDULE TASK IT4, CALLING SUBQ (A,B,I+4)
```

WAIT FOR Statements

The WAIT FOR statements cause the routine to wait for a specified scheduled subroutine, any scheduled subroutine, or all scheduled subroutines, to finish processing. One of the following forms of the WAIT FOR statements must be used with the SCHEDULE statement.

Syntax

WAIT FOR TASK *ptaskid*

ptaskid

is an integer variable or an array element of length 4, or an integer constant.

The value you provide for *ptaskid* must be a valid originated task identifier of a scheduled task; the scheduled task must finish processing before the statement following the WAIT FOR TASK statement can be processed.

Syntax

WAIT FOR ANY TASK *rtaskid*

rtaskid

is an integer variable or an array element of length 4. The value returned for *rtaskid* is the identifier of a scheduled task that has finished processing.

Each WAIT FOR ANY TASK statement detects a unique originated task completion.

Syntax

WAIT FOR ALL TASKS

The WAIT FOR ALL TASKS statement causes the scheduling routine to wait until all of its scheduled tasks have finished processing.

Example 1:

```
WAIT FOR TASK MYTASK
```

Example 2:

```
WAIT FOR ANY TASK THE_TASK
```

Example 3:

```
WAIT FOR ALL TASKS
```

TERMINATE Statement

A TERMINATE statement deletes an originated task.

Syntax

```
TERMINATE TASK taskid
```

taskid

is an integer variable or an integer array element of length 4, or an integer constant whose value is a valid originated task identifier.

The TERMINATE statement can only be used for originated tasks which are owned by the same parallel task under which the TERMINATE statement is issued. An error is detected at execution time if the TERMINATE statement is used for an originated task not owned by the parallel task, or if the originated task does not exist. An originated task cannot be terminated if it has outstanding work assigned.

When multiple levels of parallel tasks are used, the TERMINATE statement will also terminate all originated tasks owned by the originated task being terminated.

Example of Parallel Task Management Statements

```
.  
.  
INTEGER A  
ORIGINATE ANY TASK A  
.  
.  
SCHEDULE TASK A,  
+ CALLING MYSUB(1),  
+ SHARING(MINE)  
.  
.  
WAIT FOR TASK A  
.  
.  
TERMINATE TASK A
```

Figure 31. Parallel Task Management Statements Example

Parallel Loop Statements

Parallel loop statements are used to code explicit PARALLEL DO loops within your programs. When you use explicit PARALLEL DO loops, you are responsible for determining that the code you specify in the loops is valid and suitable for parallel processing.

The parallel loop statements are:

- PARALLEL DO
- LOCAL (for loops)
- DOBEFORE
- DOEVERY
- DOAFTER
- EXIT

PARALLEL DO Statement

The PARALLEL DO statement is similar to the DO statement, except that:

- Iterations of the loop can be processed concurrently and in any order.
- A PARALLEL DO permits the user to explicitly state the parallelism of the loop.

Syntax

```
PARALLEL DO [stl [,]] i = e1, e2 [, e3]
```

stl is the statement label of an executable statement, in the same program unit as the PARALLEL DO statement, that denotes the end of the PARALLEL DO loop. If you do not specify *stl*, you must use an END DO statement to indicate the end of the 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, DO
- INCLUDE
- STOP, RETURN, END
- DOBEFORE, DOEVERY, DOAFTER
- PARALLEL SECTIONS, SECTION, END SECTIONS
- Another PARALLEL DO statement
- Logical IF with an unconditional or assigned GOTO trailer statement.

i is an integer variable of length 4 called the iteration variable or PARALLEL DO variable. The iteration variable is private for the parallel loop.

When a PARALLEL DO loop is nested within a parallel loop or section, the PARALLEL DO variable automatically becomes a private variable within the range of the containing parallel loop or section. As private variables, these PARALLEL DO variables are undefined upon exit from the parallel loop or section.

If “i” appears as an argument to a subroutine or function reference, it is assumed to be unchanged by that reference. It is your responsibility to ensure this is true.

Parallel Loop Statements

e1, *e2*, and *e3*

are integer, real, or double-precision arithmetic expressions that define the values for the parallel loop iteration. *e1* is the initial value; *e2* is the terminal value; *e3* is the increment value. The increment value, *e3*, cannot have a value of 0; if it is omitted, the default value is 1.

A parallel loop cannot contain the following:

- ORIGINATE statements
- SCHEDULE statements
- TERMINATE statements
- WAIT FOR statements
- PARALLEL CALL statements
- STOP statements
- RETURN statements
- Calls to event service routines
- Character function references, including CHAR
- Character expressions (concatenation and parenthesized character arguments).

Note: Subroutines and functions referenced in a parallel loop cannot contain any of the constructs listed above, other than the RETURN statement, character function references, and character expressions.

The following also are not allowed:

- Branches to statements inside the PARALLEL DO loop by statements outside the loop.
- Branches to statements outside the PARALLEL DO loop by statements inside the loop.

For information on PARALLEL DO loop iterations and nesting of loops, see “DO and DO WHILE Statements” on page 86.

LOCAL Statement (for Parallel Loops)

The LOCAL statement for parallel loops specifies that a private instance of each variable and array listed is provided to each virtual processor participating in the execution of the parallel loop. The variables and arrays are shared for constructs nested within the parallel loop to which they are private.

Syntax

LOCAL *var* [,*var*] ...

var

specifies an arithmetic, logical, or pointer variable or array name. (Array elements and the DO variable of a containing loop cannot be specified.)

If *var* is a dummy array, its array declarator must be constant. *var* cannot be a pointee variable or array.

Variables and arrays specified by the LOCAL statement must be set within the parallel loop before being used. They become undefined as the loop ends.

Variables and arrays defined before the PARALLEL DO statement that also appear on the LOCAL statement retain the values defined prior to the PARALLEL DO statement; these values are not available inside the loop.

If the LOCAL statement for parallel loops is used, it must be the first statement (or statements) after the PARALLEL DO statement. More than one LOCAL statement can be used per loop.

When a DO or PARALLEL DO loop is nested within a parallel loop, the DO or PARALLEL DO variable is automatically a private variable within the range of the containing parallel loop and need not be specified on a LOCAL statement. As private variables, these DO and PARALLEL DO variables are undefined upon exit from the parallel loop.

When a pointer name appears on the LOCAL statement and its associated pointee variable or array is referenced within the parallel loop, the value of the private instance of the pointer will be used to address the pointee.

Private variables with the same names as variables used in array declarators are not used to determine the dimensionality of dynamically dimensioned arrays, regardless of the setting of the DDIM option.

DOBEFORE Statement

The DOBEFORE statement indicates the beginning of a block of statements that each virtual processor that participates in the execution of the PARALLEL DO will process *before* the loop is run.

Syntax

DOBEFORE [LOCK]

LOCK

specifies that only one virtual processor at a time can run the statements in the DOBEFORE block for the parallel construct.

Note: The lock is private to the parallel construct.

The DOBEFORE block is optional, and if used, must be the first block in a parallel loop.

The DOEVERY block is executed when the loop is assigned to each virtual processor. Thus, it may be executed even when the virtual processor does not execute the DOEVERY block.

Parallel Loop Statements

The following are not allowed:

- Using more than one DOBEFORE block in a parallel loop.
- Redefining the PARALLEL DO variable *i* within the DOBEFORE block. Referencing *i* is allowed, but results are not predictable.
- Branches to statements inside the DOBEFORE block by statements outside the block.
- Branches to statements outside the DOBEFORE block by statements inside the block.
- EXIT statement inside the DOBEFORE block.

DOEVERY Statement

The DOEVERY statement indicates the beginning of a block of statements whose processing is shared by the virtual processors assigned to the loop. Each virtual processor runs different iterations of the loop until all of the iterations have run.

Syntax

DOEVERY

The DOEVERY statement can be omitted if the DOEVERY block is the first block in the parallel loop.

The following are not allowed:

- Using more than one DOEVERY block in a parallel loop.
- Branches to statements inside the DOEVERY block by statements outside the block.
- Branches to statements outside the DOEVERY block by statements inside the block.

DOAFTER Statement

The DOAFTER statement indicates the beginning of a block of statements that each virtual processor that participates in the execution of the PARALLEL DO will execute *after* the loop is run.

Syntax

DOAFTER [LOCK]

LOCK

specifies that only one virtual processor at a time can run the statements in the DOAFTER block for the parallel construct.

Note: The lock is private to the parallel construct.

The DOAFTER block is optional, and if used, must be the last block in the parallel loop.

The DOAFTER block is executed when each virtual processor assigned to the loop completes processing. Thus, it may be executed even when the virtual processor did not execute the DOEVERY block.

The following are not allowed:

- Using more than one DOAFTER block in a parallel loop.
- Redefining the PARALLEL DO variable *i* within the DOAFTER block. Referencing *i* is allowed, but results are not predictable.
- Branches to statements inside the DOAFTER block by statements outside the block.
- Branches to statements outside the DOAFTER block by statements inside the block.
- EXIT statement inside the DOAFTER block.

EXIT Statement

The EXIT statement identifies when no more PARALLEL DO iterations need to be started. Any iterations of the PARALLEL DO which have been started will be completed.

Syntax

EXIT *stl*

stl is the statement label of the PARALLEL DO statement of the parallel loop whose DOEVERY block is to be halted.

The EXIT statement is provided to prevent unnecessary computation. However, its operation is not precise and may vary between executions of the program and between different executions of the same loop within the same program execution. As a result, the program logic should not be dependent on the EXIT statement preventing the execution of subsequent iterations of the loop.

```

      .
      .
10  PARALLEL DO I=1, N
      PARALLEL DO J=1, M
      LOCAL D1
      D1=SQRT(X(I,J)**2 + (Y(I,J)**2))
      IF (D1.GT.TVAL) THEN
        FVAL=D1
        EXIT 10
      ENDIF
      ENDDO
    ENDDO
      .
      .
    END
  
```

Figure 32. EXIT Statement Example

Example of Parallel Loop Statements

The following example uses a parallel loop to find the sum of the elements of a matrix. Partial sums of the elements can be found in parallel for each column of the matrix M. PTOT is a private variable for each processor to accumulate the partial sums in. As each processor completes its work, the partial sums are accumulated in the DOAFTER block to determine the sum.

Note that the DOAFTER block has LOCK specified to ensure that only one virtual processor updates the shared variable SUM at a time.

```
@PROCESS PAR(LANG)
  SUBROUTINE PLOOP(SUM)
    REAL M(500,500)
    SUM=0.0
    PARALLEL DO I = 1, 500
      LOCAL PTOT
      DOBEFORE
        PTOT = 0.0
      DOEVERY
        DO J = 1, 500
          PTOT = PTOT + M(J,I)
        END DO
      DOAFTER LOCK
        SUM = SUM + PTOT
      END DO
    RETURN
  END
```

Figure 33. Parallel Loop Statements Example

Compile Time Serializations: If a PARALLEL DO loop is serialized, first the DOBEFORE block (if any) will be run, then the DOEVERY block will be run as a loop, and lastly the DOAFTER block (if any) will be run. Variables and arrays specified on the LOCAL statement and loop induction variables will be private during the execution of all these blocks, and will thus be undefined after the last block has been executed. An EXIT statement will cause a branch to the first statement of the DOAFTER block, (if there is a DOAFTER block), or to the first statement after the DOEVERY block (if there is no DOAFTER block).

Parallel Sections Statements

Parallel Sections statements are used to code sections of your program that are to run in parallel. When you explicitly code a section of your program to run in parallel, you are responsible for determining that the code you include in the parallel section is valid and suitable for parallel processing.

The statements used for coding parallel sections are:

- PARALLEL SECTIONS
- LOCAL (for sections)
- SECTION
- END SECTIONS

PARALLEL SECTIONS Statement

The PARALLEL SECTIONS statement indicates the beginning of a group of sections that can be run in parallel with each other.

Syntax

PARALLEL SECTIONS

Any number of sections can be specified within a group of parallel sections.

LOCAL Statement (for Parallel Sections)

The LOCAL statement for parallel sections specifies that an instance of each variable or array listed is provided to each virtual processor that participates in the execution of the parallel sections. The variables and arrays are shared for constructs nested within the one to which they are private.

Syntax

```
LOCAL var [,var] ...
```

var

specifies an arithmetic, logical, or pointer variable or array name. (Array elements and the DO variable of a containing loop cannot be specified.)

If *var* is a dummy array, its array declarator must be constant. *var* may not be a pointee variable or array.

Variables and arrays specified by the LOCAL statement must be set within the parallel section before being used. They become undefined as the section ends.

Variables and arrays defined before the PARALLEL SECTIONS statement that also appear on the LOCAL statement retain the values defined prior to the PARALLEL SECTIONS statement; these values are not available inside the sections.

If the LOCAL statement for parallel sections is used, it must be the first statement (or statements) after the PARALLEL SECTIONS statement. More than one LOCAL statement can be used in a group.

When a DO or PARALLEL DO loop is nested within a parallel section, the DO or PARALLEL DO variable is automatically a private variable within the range of the containing parallel sections and need not be specified on a LOCAL statement. As private variables, these DO and PARALLEL DO variables are undefined upon exit from the parallel section.

When a pointer name appears on the LOCAL statement and its associated pointee variable or array is referenced within the parallel section, the value of the private instance of the pointer will be used to address the pointee.

Private variables with the same names as variables used in array declarators are not used to determine the dimensionality of dynamically dimensioned arrays, regardless of the setting of the DDIM option.

SECTION Statement

The SECTION statement indicates the beginning of a block of statements, which will be processed as a parallel thread.

Syntax

```
SECTION [m][,WAITING (n1 [,n2] ...)]
```

m is a positive integer constant up to 5 digits in length that optionally identifies the section.

WAITING (*n1* [,*n2*] ...)

specifies that this section is not to be started until after the sections *n1*,*n2*... have completed. The maximum number of sections that can be waited on in a group of parallel sections is 1023.

n is a positive integer constant up to 5 digits in length that identifies a section for which this section is to wait. *n* must refer to a previously specified section in the current group of parallel sections.

A section is ended by either another SECTION statement or by an END SECTIONS statement. If any section within a PARALLEL SECTIONS construct is determined to be unsuitable for parallel processing, all the included sections are serialized.

A section cannot contain the following:

- ORIGINATE statements
- TERMINATE statements
- SCHEDULE statements
- WAIT FOR statements
- PARALLEL CALL statements
- STOP statements
- RETURN statements
- Calls to event service routines
- Character function references, including CHAR
- Character expressions (concatenation and parenthesized character arguments).

Note: Subroutines and functions referenced in a parallel section cannot contain any of the constructs listed above, other than the RETURN statement, character function references, and character expressions.

The following are also not allowed:

- Branches to statements inside the SECTION block by statements outside the block.
- Branches to statements outside the SECTION block by statements inside the block.

END SECTIONS Statement

The END SECTIONS statement terminates a group of parallel sections.

Syntax

END SECTIONS

Example of Parallel Sections Statements

```

@PROCESS PAR(LANG) OPT(3)
  SUBROUTINE PSECT(TOT)
  EXTERNAL SUMFUN,TOTFUN
  COMMON /TCOM/ S(500),T(500)
  PARALLEL SECTIONS
  LOCAL SUMVAL
  SECTION 1
    SUMVAL = SUMFUN(1,250)
    SUM1 = SUMVAL**2
  SECTION 2
    SUMVAL = SUMFUN(251,500)
    SUM2 = SUMVAL**3
  SECTION 3, WAITING(1,2)
    SUM3 = TOTFUN(SUM1,SUM2)
  SECTION 4
    CALL SUB1(SUM4)
  END SECTIONS
  TOT = SUM1+SUM2+SUM3+SUM4
  RETURN
  END

```

Figure 34. Parallel Sections Statements Example

Parallel Call Statements

You can use the parallel call statements to call subroutines that run concurrently with the calling routine, and to coordinate the completion of the called subroutines.

The parallel call statements are:

- PARALLEL CALL
- WAIT FOR ALL CALLS

PARALLEL CALL Statement

The PARALLEL CALL statement assigns a subroutine to run as a parallel thread. The PARALLEL CALL statement is similar to the CALL statement, except that the statement following PARALLEL CALL can proceed without waiting for the subroutine to complete. The statements following subsequent PARALLEL CALL statements also proceed without waiting for subroutines already activated by previous PARALLEL CALL statements to complete.

Syntax

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

name

is the name of a subroutine or entry point. This name can be a dummy argument.

arg

is an argument that is supplied to the subroutine. The argument can be a variable, array element, array name, constant, or expression (including a parenthesized primary). The argument cannot be a function name, a subroutine name, or an asterisk (*) followed by a statement label.

Arguments that are expressions or current DO variables are passed by copy. Data passed in common blocks that are specified in the program unit containing the parallel call are also shared. (See “Common Blocks in Parallel Programs” on page 74.) In order to preserve computational independence, the same variables or array elements should not be referenced by the calling routine until a WAIT FOR ALL CALLS statement is specified, unless the appropriate control mechanisms are used.

Subroutines specified on the PARALLEL CALL statement cannot contain:

- ORIGINATE statements
- TERMINATE statements
- SCHEDULE statements
- WAIT FOR statements
- Calls to event service routines

Note: Subroutines and functions given control in the subroutine invoked by a PARALLEL CALL statement cannot contain any of the constructs listed above.

A PARALLEL CALL statement cannot be used within a PARALLEL DO or PARALLEL SECTIONS construct.

For information on calling conventions, see “CALL Statement” on page 67.

Example 1:

```
PARALLEL CALL SUBA
```

Example 2:

```
PARALLEL CALL ANALYSIS_PGM(INTERESTING_VALUE,3,I+4,(ITER))
```

Note: I+4 and ITER will be passed by copy.

WAIT FOR ALL CALLS Statement

The WAIT FOR ALL CALLS statement causes the calling routine to wait until all subroutines invoked with PARALLEL CALL within the same parallel thread have completed processing. Running then continues with the statement following the WAIT FOR ALL CALLS statement.

Syntax

WAIT FOR ALL CALLS

There is an implicit WAIT FOR ALL CALLS at the end of the main program and at the end of a scheduled subroutine or a subroutine activated by a PARALLEL CALL.

The WAIT FOR ALL CALLS statement is independent of WAIT FOR TASK, WAIT FOR ANY TASK, and WAIT FOR ALL TASKS statements. To synchronize parallel processing that combines both scheduled subroutines and subroutines invoked with PARALLEL CALL, both the WAIT FOR ALL CALLS and the WAIT FOR (TASK | ANY TASK | ALL TASKS) statements must be coded.

Example of Parallel Call Statements

```

        DIMENSION A(4,100,100)
        COMMON/share/B(4,100)
        .
        .
        DO 10 I=1,3
            PARALLEL CALL SUBWRK(I,A(I,1,1) )
10    CONTINUE
        CALL SUBWRK(4,A(4,1,1))
        WAIT FOR ALL CALLS
        .
        .
        END

        SUBROUTINE SUBWRK(I,ARIN)
        DIMENSION ARIN(4,100,100)
        COMMON/share/B(4,100)
        COMMON/work/C(100,100)
        .
        .
        RETURN
        END

```

Figure 35. Parallel Call Statements Example

Chapter 6. Intrinsic Functions

Some intrinsic functions can be referenced by a *specific name*, some by a *generic name*, and some by both. In this chapter, a function described as a specific name requires a specific argument type and produces a result of a specific type. A generic name does not require a specific argument type and usually produces a result of the same type as that of the argument. Generic names simplify the referencing of intrinsic functions.

Intrinsic Function Table

The categories of intrinsic functions are:

- Exponential and logarithmic
- Trigonometric
- Hyperbolic
- Miscellaneous mathematical
- Conversion and maximum/minimum
- Storage Functions
- Character manipulation
- Bit manipulation.

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 subprogram unit. (See “INTRINSIC Statement” on page 163.)

References to the functions are resolved from the library, inserted into the object module, or evaluated by the compiler. That is, the code generated by VS FORTRAN Version 2 for the reference contains instructions to link to the function in the library (out-of-line), instructions to perform the function directly (in-line), or the actual value of the function.

For a small subset of the mathematical functions, alternative functions are available that provide results compatible with VS FORTRAN Version 1. These functions are identified by note 3 and are identified in “Alternative Mathematical Library Subroutines” on page 317.

Figure 36 (Page 1 of 10). Intrinsic Functions

Intrinsic Function	Generic Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argument	Type and Range of Function
<i>Exponential and Logarithmic Functions</i>							
Exponential	EXP	EXP(3) DEXP(3) QEXP CEXP	$y = e(x)$ $y = e(z)$	$x \leq 174.673$ $-180.216 \leq x \leq 174.673$ $xsub1 \leq 174.673; xsub2 < (2^{**}18 * \pi)$ $xsub1 \leq 174.673; xsub2 < (2^{**}50 * \pi)$ $xsub1 \leq 174.673; xsub2 < 2^{**}100$	1	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*32	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*32
Natural logarithm	LOG	ALOG(3)	$y = \text{logsube}(x)$	$x > 0.0$	1	REAL*4	REAL*4 $y \geq -180.218$ $y \leq 174.673$ REAL*8 $y \geq -180.218$ $y \leq 174.673$ REAL*16 $y \geq -180.218$ $y \leq 174.673$ COMPLEX*8 $ysub1 \geq -180.218$ $ysub1 \leq 175.021$ $-\pi \leq ysub2 \leq \pi$ COMPLEX*16 $ysub1 \geq -180.218$ $ysub1 \leq 175.021$ $-\pi \leq ysub2 \leq \pi$ COMPLEX*32 $ysub1 \geq -180.218$ $ysub1 \leq 175.021$ $-\pi \leq ysub2 \leq \pi$
Common logarithm	LOG10	ALOG10(3) DLOG10(3) QLOG10	$y = \text{logsub10}(x)$	$x > 0.0$	1	REAL*4 REAL*8 REAL*16	REAL*4 $y \geq -78.268$ $y \leq 75.859$ REAL*8 $y \geq -78.268$ $y \leq 75.859$ REAL*16 $y \geq -78.268$ $y \leq 75.859$

Figure 36 (Page 2 of 10). Intrinsic Functions

Intrinsic Function	Generic Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argument	Type and Range of Function
<i>Trigonometric Functions</i>							
Sine	SIN	SIN(3) DSIN(3) QDIN(3) CDSIN CQDIN	$y = \sin(x)$ $y = \sin(z)$	$ x < (2^{**18} * \pi)$ $ x < (2^{**50} * \pi)$ $ x < 2^{**100}$ $ xsub1 < (2^{**18} * \pi); xsub2 \leq 174.673$ $ xsub1 < (2^{**50} * \pi); xsub2 \leq 174.673$ $ xsub1 < 2^{**100}; xsub2 \leq 174.673$	1	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*32	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*32
Cosine	COS	COS(3) DCOS(3) QCOS CCOS CDCOS CQCOS	$y = \cos(x)$ $y = \cos(z)$	$ x < (2^{**18} * \pi)$ $ x < (2^{**50} * \pi)$ $ x < 2^{**100}$ $ xsub1 < (2^{**18} * \pi); xsub2 \leq 174.673$ $ xsub1 < (2^{**50} * \pi); xsub2 \leq 174.673$ $ xsub1 < 2^{**100}; xsub2 \leq 174.673$	1	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*32	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*32
Tangent	TAN	TAN(3,5) DTAN(3,5) QTAN(4)	$y = \tan(x)$	$ x < (2^{**18} * \pi)$ $ x < (2^{**50} * \pi)$ $ x < 2^{**100}$	1	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Cotangent	COTAN	COTAN(3,5) DCOTAN(3,5) QCOTAN(4)	$y = \cot(x)$	$ x < (2^{**18} * \pi)$ $ x < (2^{**50} * \pi)$ $ x < 2^{**100}; x > 16^{**(-63)}$	1	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arcsine	ASIN	ASIN(3) DASIN(3) QARSIN	$y = \arcsin(x)$	$ x \leq 1$	1	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arccosine	ACOS	ACOS(3) DACOS(3) QARCOS	$y = \arccos(x)$	$ x \leq 1$	1	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16

Figure 36 (Page 3 of 10). Intrinsic Functions

Figure 36 (Page 3 of 10). Intrinsic Functions								
Intrinsic Function	Gen- eric Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argu- ment	Type and Range of Function	
Arctangent	ATAN	ATAN(3)	$y = \arctan(x)$	any REAL*4 argument	1	REAL*4	REAL*4 $-\pi/2 \leq y \leq \pi/2$	
		DATAN(3)		any REAL*8 argument		REAL*8	REAL*8 $-\pi/2 \leq y \leq \pi/2$	
		QATAN		any REAL*16 argument	2	REAL*16	REAL*16 $-\pi/2 \leq y \leq \pi/2$	
		ATAN2	ATAN2(3)	Computes angle y such that $\sin(y) = x_{\text{sub1}} / ((x_{\text{sub1}}^{**2} + (x_{\text{sub2}})^{**2})^{**<1/2>})$ $\cos(y) = x_{\text{sub2}} / ((x_{\text{sub1}})^{**2} + (x_{\text{sub2}})^{**2})^{**<1/2>})$	any REAL*4 arguments (except 0, 0) any REAL*8 arguments (except 0, 0) any REAL*16 arguments (except 0, 0)	REAL*4 REAL*8 REAL*16	REAL*4 $-\pi < y \leq \pi$ REAL*8 $-\pi < y \leq \pi$ REAL*16 $-\pi < y \leq \pi$	
Hyperbolic Functions								
Hyperbolic sine	SINH	SINH(3) DSINH QSINH	$y = (e^{**x} - e^{**(-x)}) / 2$	$ x < 175.366$	1	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16	
	Hyperbolic cosine	COSH	COSH(3)	$y = (e^{**x} + e^{**(-x)}) / 2$	$ x < 175.366$	1	REAL*4 REAL*8 REAL*16	REAL*4 $1 \leq y \leq \Omega$ REAL*8 $1 \leq y \leq \Omega$ REAL*16 $1 \leq y \leq \Omega$
			DCOSH QCOSH					
Hyperbolic tangent	TANH	TANH(3) DTANH QTANH	$y = (e^{**x} - e^{**(-x)}) / (e^{**x} + e^{**(-x)})$	any REAL*4 argument any REAL*8 argument any REAL*16 argument	1	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16	
	Miscellaneous Mathematical Functions							
	Truncation	AINT	AINT DINT QINT	$y = \text{sgn}(x) \bullet [x]$	any REAL*4 argument any REAL*8 argument any REAL*16 argument	1	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16

Figure 36 (Page 4 of 10). Intrinsic Functions

Intrinsic Function	Generic Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argument	Type and Range of Function
Nearest whole number	ANINT	ANINT DNINT	$y = \text{sgn}(x) \cdot \lceil x+5 \rceil$ if $x \geq 0$ $y = \text{sgn}(x) \cdot \lfloor x-5 \rfloor$ if $x < 0$	any REAL*4 argument any REAL*8 argument	1	REAL*4 REAL*8	REAL*4 REAL*8
Nearest integer	NINT	NINT IDNINT	$y = \text{sgn}(x) \cdot \lceil x+5 \rceil$ if $x \geq 0$ $y = \text{sgn}(x) \cdot \lfloor x-5 \rfloor$ if $x < 0$	any REAL*4 argument any REAL*8 argument	1	REAL*4 REAL*8	INTEGER*4 INTEGER*4
Absolute value	ABS	ABS(1) ⁹ ABS DABS QABS CABS(3,6) CDABS(3,6) CQABS(6)	$y = x $ $y = z = (\text{re}(z)**2 + \text{im}(z)**2) ** (1/2)$	any integer argument any REAL*4 argument any REAL*8 argument any REAL*16 argument any COMPLEX*8 argument any COMPLEX*16 argument any COMPLEX*32 argument	1	any integer REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*32	same as x REAL*4 REAL*8 REAL*16 REAL*4 REAL*8 REAL*16
Error function	ERF	ERF	$y = 2 / \sqrt{\pi} \int_0^x e^{-u^2} du$	any REAL*4 argument any REAL*8 argument any REAL*16 argument	1	REAL*4 REAL*8 REAL*16	REAL*4 -1 ≤ y ≤ 1 REAL*8 -1 ≤ y ≤ 1 REAL*16 -1 ≤ y ≤ 1
	ERFC	ERFC	$y = 2 / \sqrt{\pi} \int_x^\infty e^{-u^2} du$	any REAL*4 argument		REAL*4	REAL*4 0 ≤ y ≤ 2
	DERFC	DERFC	integral from x to infinity of $e^{-u^2} du$	any REAL*8 argument		REAL*8	REAL*8 0 ≤ y ≤ 2
	QERFC	QERFC	$y = 1 - \text{ERF}(x)$	any REAL*16 argument		REAL*16	REAL*16 0 ≤ y ≤ 2

Figure 36 (Page 5 of 10). Intrinsic Functions

Intrinsic Function	Generic Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argument	Type and Range of Function
Gamma and log gamma		GAMMAGAMMA	$y = \text{integral from } 0 \text{ to } \infty \text{ of } < u^{**}(x-1) e^{**}(-u) du >$	$2^{**}(-252) < x < 57.5744$	1	REAL*4	REAL*4
		DGAMMA				REAL*8	0.88560 $\leq y \leq \Omega$ REAL*8
		LGAMMALGAMMA		$0 < x < 4.2913 \bullet 10^{**73}$		REAL*4	0.88560 $\leq y \leq \Omega$ REAL*4
		DLGAMA	$y = \text{logsube}(\text{gamma}(x))$ or $y = \text{logsube integral from } 0 \text{ to } \infty \text{ of } < u^{**}(x-1) e^{**}(-u) du >$			REAL*8	-0.12149 $\leq y \leq \Omega$ REAL*8 - 0.12149 $\leq y \leq \Omega$
Remaindering	MOD	MOD(7,10) AMOD(7) DMOD(7) QMOD(7)	$y = \text{xsub1} - [\text{xsub1}/\text{xsub2}] * \text{xsub2}$	$\text{xsub2} \neq 0$	2	any integer REAL*4 REAL*8 REAL*16	same as xsub1 REAL*4 REAL*8 REAL*16
Transfer of sign	SIGN	ISIGN(1) ^o SIGN DSIGN QSIGN	$y = \text{sgn}(\text{xsub2}) * \text{xsub1} $	any integer argument any REAL*4 argument any REAL*8 argument any REAL*16 argument	2	any integer REAL*4 REAL*8 REAL*16	same as xsub1 REAL*4 REAL*8 REAL*16
Positive difference	DIM	IDIM(1) ^o DIM DDIM QDIM	$y = \text{xsub1} - \text{xsub2}$ if $\text{xsub1} > \text{xsub2}$ $y = 0$ if $\text{xsub1} \leq \text{xsub2}$	any integer argument any REAL*4 argument any REAL*8 argument any REAL*16 argument	2	any integer REAL*4 REAL*8 REAL*16	same as xsub1 REAL*4 REAL*8 REAL*16
Double precision product		DPROD	$y = \text{xsub1} \bullet \text{xsub2}$	any REAL*4 argument	2	REAL*4	REAL*8
Imaginary part of a complex argument	IMAG	AIMAG DIMAG QIMAG	$y = \text{im}(z)$	any COMPLEX*8 argument any COMPLEX*16 argument any COMPLEX*32 argument	1	COMPLEX*8 COMPLEX*16 COMPLEX*32	REAL*4 REAL*8 REAL*16
Complex conjugate	CONJUG	CONJUG DCONJUG QCONJUG	$y = \text{re}(z) - \text{im}(z)$	any COMPLEX*8 argument any COMPLEX*16 argument any COMPLEX*32 argument	1	COMPLEX*8 COMPLEX*16 COMPLEX*32	COMPLEX*8 COMPLEX*16 COMPLEX*32

Figure 36 (Page 6 of 10). Intrinsic Functions

Intrinsic Function	Generic Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argument	Type and Range of Function
Square root	SQRT	SQRT(3) DSQRT(3) QSQRT CSQRT	$y = x^{**}(1/2)$, $x \geq 0.0$ $y = z^{**}(1/2)$	any REAL*4 argument ≥ 0.0 any REAL*8 argument ≥ 0.0 any REAL*16 argument ≥ 0.0 any COMPLEX*8 argument	1	REAL*4 REAL*8 REAL*16 COMPLEX*8	REAL*4 REAL*8 REAL*16 COMPLEX*8 $0 \leq \text{ysub1} \leq 1.0987 * (\Omega^{**}(1/2))$) $ \text{ysub2} \leq 1.0987 * (\Omega^{**}(1/2))$ COMPLEX*16 $0 \leq \text{ysub1} \leq 1.0987 * (\Omega^{**}(1/2))$) $ \text{ysub2} \leq 1.0987 * (\Omega^{**}(1/2))$ COMPLEX*32 $0 \leq \text{ysub1} \leq 1.0987 * (\Omega^{**}(1/2))$) $\text{ysub2} \leq 1.0987 * (\Omega^{**}(1/2))$
		CDSQRT		any COMPLEX*16 argument		COMPLEX*16	
		CQSQRT		any COMPLEX*32 argument		COMPLEX*32	
Conversion and Maximum/Minimum Functions(2)							
Conversion to type integer	INT	— IFIX IDINT IQINT — HFIX(8)	$y = \text{sgn}(x) * [x]$ $y = \text{int}(\text{re}(z))$ $y = \text{sgn}(x) * [x]$	any integer argument any REAL*4 argument any REAL*8 argument any REAL*16 argument any COMPLEX*8 argument	1	any integer REAL*4 REAL*8 REAL*16 COMPLEX*8 REAL*4	same as x INTEGER*4 INTEGER*4 INTEGER*4 INTEGER*4 INTEGER*2
Conversion to type real	REAL	FLOAT(1) ^o — SINGL SINGLQ — DREAL QREAL	$y = x$ $y = \text{re}(z)$	any integer argument any REAL*4 argument any REAL*8 argument any REAL*16 argument any COMPLEX*8 argument any COMPLEX*16 argument any COMPLEX*32 argument	1	any integer REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*32	REAL*4 REAL*4 REAL*4 REAL*4 REAL*4 REAL*8 REAL*16
Conversion to type double precision	DBLE	DFLOAT(1) ^o DBLE — DBLEQ —	$y = x$ $y = \text{re}(z)$	any integer argument any REAL*4 argument any REAL*8 argument any REAL*16 argument any COMPLEX*8 argument	1	any integer REAL*4 REAL*8 REAL*16 COMPLEX*8	REAL*8 REAL*8 REAL*8 REAL*8 REAL*8

Figure 36 (Page 7 of 10). Intrinsic Functions

Intrinsic Function	Generic Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argument	Type and Range of Function
Conversion to type extended precision	QEXT	QFLOAT(1) ⁹ QEXT QEXTD	$y = x$	any integer argument any REAL*4 argument any REAL*8 argument	1	any integer REAL*4 REAL*8	REAL*16 REAL*16 REAL*16
Conversion to type complex	CMPLX	— CMPLX — QCMPLEX — — DCMPLEX(8)	$y = x + 0i$, one argument $y = xsub1 + xsub2i$, two arguments $y = z$ (1 argument only) $y = x + 0i$, one argument $y = xsub1 + xsub2i$, two arguments	any integer argument any REAL*4 argument any REAL*8 argument any REAL*16 argument any COMPLEX*8 argument any REAL*8 argument	1 or 2	any integer REAL*4 REAL*8 REAL*16 COMPLEX*8 REAL*8	COMPLEX*8 COMPLEX*8 COMPLEX*8 COMPLEX*32 COMPLEX*8 _____ COMPLEX*16
Choosing largest value	MAX	MAX0(1) ⁹ AMAX1 DMAX1 QMAX1 — — AMAX0(8,10) MAX1(8)	$y = \max(xsub1, \dots, xsubn)$	any integer arguments any REAL*4 arguments any REAL*8 arguments any REAL*16 arguments any integer arguments any REAL*4 arguments	≥ 2	any integer REAL*4 REAL*8 REAL*16 _____ any integer REAL*4	same as xsub1 REAL*4 REAL*8 REAL*16 _____ REAL*4 INTEGER*4
Choosing smallest value	MIN	MIN0(1) ⁹ AMIN1 DMIN1 QMIN1 — — AMIN0(8,10) MIN1(8)	$y = \min(xsub1, \dots, xsubn)$	any integer arguments any REAL*4 arguments any REAL*8 arguments any REAL*16 arguments any integer arguments any REAL*4 arguments	≥ 2	any integer REAL*4 REAL*8 REAL*16 _____ any integer REAL*4	same as xsub1 REAL*4 REAL*8 REAL*16 _____ REAL*4 INTEGER*4
Storage Functions							

Figure 36 (Page 8 of 10). Intrinsic Functions

Intrinsic Function	Generic Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argument	Type and Range of Function
Allocation status		ALLOCATED	TRUE if storage is allocated for pointee or pointer addresses dynamically allocated storage; otherwise FALSE	any	1	any	LOGICAL*4
Location of variable		LOC	Address of argument	any	1	any	POINTER
Character Manipulation Functions							
Conversion to type integer		ICHAR	Position of x in collating sequence	any CHARACTER*1 argument	1	CHARACTER*1	INTEGER*4
Conversion to type character		CHAR(1) ^o	Character corresponding to position of x in collating sequence	any integer argument	1	any integer	CHARACTER*1
Length		LEN	Length of x	any CHARACTER argument	1	CHARACTER	INTEGER*4
Index of a substring		INDEX	Location of substring xsub2 in string xsub1	any CHARACTER arguments	2	CHARACTER	INTEGER*4
Lexically greater than or equal		LGE	xsub1 ≥ xsub2 Comparison is ASCII	any CHARACTER arguments	2	CHARACTER	LOGICAL*4
Lexically greater than		LGT	xsub1 > xsub2 Comparison is ASCII	any CHARACTER arguments	2	CHARACTER	LOGICAL*4
Lexically less than or equal		LLE	xsub1 ≤ xsub2 Comparison is ASCII	any CHARACTER arguments	2	CHARACTER	LOGICAL*4
Lexically less than		LLT	xsub1 < xsub2 Comparison is ASCII	any CHARACTER arguments	2	CHARACTER	LOGICAL*4
Bit Manipulation Functions							
Inclusive or		IOR(1) ^o	y = or(xsub1,xsub2)	any integer or unsigned arguments	2	any integer or unsigned	same as xsub1
Logical and		IAND(1) ^o	y = and(xsub1,xsub2)	any integer or unsigned arguments	2	any integer or unsigned	same as xsub1
Logical complement		NOT(1) ^o	y = not(xsub1)	any integer or unsigned argument	1	any integer or unsigned	same as xsub1 *****

Figure 36 (Page 9 of 10). Intrinsic Functions

Intrinsic Function	Generic Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argument	Type and Range of Function
Exclusive or		IEOR(1) ^o XOR(1) ^o	$y = \text{xor}(\text{xsub1}, \text{xsub2})$	any integer or unsigned arguments	2	any integer or unsigned	same as xsub1
Shift bits		LSHIFT(1) ^o	$y = \text{shift}(\text{xsub1}, \text{xsub2})$ xsub1 is shifted by xsub2 bits to right if xsub2 < 0 or to left if xsub2 > 0 Vacated bits are filled with zeros	xsub1 is any integer or unsigned argument; $-8 \leq \text{xsub2} \leq 8$ $-16 \leq \text{xsub2} \leq 16$ $-32 \leq \text{xsub2} \leq 32$ $-64 \leq \text{xsub2} \leq 64$	2	any integer or unsigned INTEGER*1 or UnSigned*1 INTEGER*2 INTEGER*4 INTEGER*8	same as xsub1
Shift left		LSHIFT(1) ^o	$y = \text{shift_left}(\text{xsub1}, \text{xsub2})$ xsub1 is shifted left by xsub2 bits Vacated bits are filled with zeros	xsub1 is any integer or unsigned argument; $0 \leq \text{xsub2} \leq 8$ $0 \leq \text{xsub2} \leq 16$ $0 \leq \text{xsub2} \leq 32$ $0 \leq \text{xsub2} \leq 64$	2	any integer or unsigned INTEGER*1 or UnSigned*1 INTEGER*2 INTEGER*4 INTEGER*8	same as xsub1
Shift right		RSHIFT(1) ^o	$y = \text{shift_right}(\text{xsub1}, \text{xsub2})$ xsub1 is shifted right by xsub2 bits Vacated bits are filled with zeroes	xsub1 is any integer or unsigned argument; $0 \leq \text{xsub2} \leq 8$ $0 \leq \text{xsub2} \leq 16$ $0 \leq \text{xsub2} \leq 32$ $0 \leq \text{xsub2} \leq 64$	2	any integer or unsigned INTEGER*1 or UnSigned*1 INTEGER*2 INTEGER*4 INTEGER*8 c.same as xsub1 *****	same as xsub1
Shift circularly		ISHFTC(1) ^o	$y = \text{shift_circle}(\text{xsub1}, \text{xsub2}, \text{xsub3})$ xsub1 is shifted right circularly by xsub2 bits If xsub3 is present, only the right-most xsub3 bits participate No bits are lost	xsub1 is any integer or unsigned argument; $0 \leq \text{xsub2} \leq 7$; $0 \leq \text{xsub3} \leq 8$ $0 \leq \text{xsub2} \leq 15$; $0 \leq \text{xsub3} \leq 16$ $0 \leq \text{xsub2} \leq 31$; $0 \leq \text{xsub3} \leq 32$ $0 \leq \text{xsub2} \leq 63$; $0 \leq \text{xsub3} \leq 64$	2 or 3	any integer or unsigned INTEGER*1 or UnSigned*1 INTEGER*2 INTEGER*4 INTEGER*8	same as xsub1

Figure 36 (Page 10 of 10). Intrinsic Functions

Intrinsic Function	Generic Name	Specific Name	Definition(1)	Argument Range (11)	No. of Args	Type of Argument	Type and Range of Function
Bit test		BTEST(9,10)	y = bittest(xsub1,xsub2) true if bit xsub2 of xsub1 = 1 false if bit xsub2 of xsub1 = 0	xsub1 is any integer or unsigned argument; 0 ≤ xsub2 ≤ 7 0 ≤ xsub2 ≤ 15 0 ≤ xsub2 ≤ 31 0 ≤ xsub2 ≤ 63	2	any integer or unsigned INTEGER*1 or UnSigned*1 INTEGER*2 INTEGER*4 INTEGER*8	LOGICAL*4
Bit set		IBSET(9,10)	y = bitset(xsub1,xsub2) Sets bit xsub2 of xsub1 to 1	xsub1 is any integer or unsigned argument; 0 ≤ xsub2 ≤ 7 0 ≤ xsub2 ≤ 15 0 ≤ xsub2 ≤ 31 0 ≤ xsub2 ≤ 63	2	any integer or unsigned INTEGER*1 or UnSigned*1 INTEGER*2 INTEGER*4 INTEGER*8	same as xsub1
Bit clear		IBCLR(9,10)	y = bitclear(xsub1,xsub2) Sets bit xsub2 of xsub1 to 0	xsub1 is any integer or unsigned argument; 0 ≤ xsub2 ≤ 7 0 ≤ xsub2 ≤ 15 0 ≤ xsub2 ≤ 31 0 ≤ xsub2 ≤ 63	2	any integer or unsigned INTEGER*1 UnSigned*1 INTEGER*2 INTEGER*4 INTEGER*8	same as xsub1
Bit extraction		IBITS(9,10)	y = bitextrat(xsub1,xsub2,xsub3) Extracts xsub3 bits of xsub1 starting with xsub2	xsub1 is any integer or unsigned argument; 0 ≤ xsub2 ≤ 7; 0 ≤ xsub3 ≤ 8 0 ≤ xsub2 ≤ 15; 0 ≤ xsub3 ≤ 16 0 ≤ xsub2 ≤ 31; 0 ≤ xsub3 ≤ 32 0 ≤ xsub2 ≤ 63; 0 ≤ xsub3 ≤ 64	3	any integer or unsigned INTEGER*1 or UnSigned*1 INTEGER*2 INTEGER*4 INTEGER*8	same as xsub1

Notes:

- (1) Definitions use familiar mathematical function names, which have their mathematical meanings or are defined below.

The result of a function of type complex is the principal value.

Meanings of symbols:

x denotes a single argument.

$x_{\text{sub}i}$ denotes the i -th argument when a function accepts more than one argument.

$[x]$ denotes the integer part of the number x .

$\text{sgn}(x)$ is $+1$ if $x \geq 0$ or -1 if $x < 0$.

y denotes a function result.

z denotes a complex argument.

$\text{re}(z)$ denotes the real part of the complex number z .

$\text{im}(z)$ denotes the imaginary part of the complex number z .

$\Omega = 16(63) (1 - 16^{**}(-6))$ for single precision routines, $16(63) (1 - 16^{**}(-14))$ for double precision routines, and $16(63) (1 - 16^{**}(-28))$ for extended precision.

- (2) The generic name must be used for conversion functions when no specific name is supplied.
- (3) Also available in the alternate mathematical library, which provides alternative functions that provide results compatible with VS FORTRAN Version 1.
- (4) x must not lie with 8 units of last place of a singularity. Singularities are $\pm (2n + 1)\pi/2$, $n = 0, 1, 2, \dots$ for tangent, and $\pm n\pi$, $n = 0, 1, 2, \dots$ for cotangent.
- (5) Multiples of π are singularities of the cotangent function. Multiples of π added to $\pi/2$ ($n * \pi + \pi/2$, n is an integer) are singularities of the tangent function.
- (6) Floating-point overflow can occur.
- (7) If $x_{\text{sub}2}=0$, the modulus function is mathematically undefined. In addition, a divide exception is recognized and an interruption occurs.
- (8) The specific name must be used to obtain a function value of this type.
- (9) The bits in bit-manipulation functions are numbered from right to left, beginning at zero.
- (10) This specific name is also a generic name when used with integer argument(s); it is specific only with INTEGER*4 argument(s). When there is more than one argument, all arguments must agree in length.
- (11) Throughout these tables, the following approximate values are represented by $(2(18) * \pi)$ and $(2^{**}(50) * \pi)$:

$$(2^{**}(18) * \pi) = .8235496645826428\text{D} + 06$$

$$(2^{**}(50) * \pi) = .3537118876014220\text{D} + 16$$

Math Function Accuracy Information

This section contains further information about the arguments of the explicitly called mathematical functions, giving the effect of an argument error upon the accuracy of the answer returned and, for some of the routines, the accuracy of the algorithm.

The information is arranged alphabetically according to its function (that is, absolute value, exponentiation, logarithmic, etc.). The specific names for each function are arranged logically from real to complex. For example, under the heading “Square Root Functions,” the information is arranged in the order SQRT, DSQRT, CSQRT, CDSQRT.

Units in the Last Place (ULPs)

Accuracy of the functions is described in terms of Units in the Last Place, or “ulp”s.

For a positive normalized floating-point number X in a floating-point number system, a Unit in the Last Place of X is defined as the difference between X and the next larger floating-point number. For example, an ulp of the short precision floating-point hexadecimal number 12345678 is 12000001, which equals 0D,100000 after normalization of the ulp value.

If x is a positive *real* number (of infinite precision) lying in the range of floating-point numbers, then an ulp of x , from the standpoint of the floating-point number system, is defined as an ulp of the largest floating-point number X that does not exceed x .

It is convenient to define an ulp of a negative floating-point or real number as the negative of an ulp of its absolute value.

The indicated routines provide results falling into two categories:

The point rule error <0.5 ULPs	The answer (as the computer can best represent it)
Two point rule error <1.0 ULPs	Either the answer, or one away (in one direction)

The accuracy of an answer provided by the routines is influenced by two factors: the performance of the subprogram and the accuracy of the argument. The effect of an argument error upon the accuracy of an answer depends solely upon the mathematical algorithm involved and not upon the particular coding used in the subprogram.

A guide to the effect of propagated argument errors is provided, because argument errors always influence the accuracy of answers. This guide (expressed as a simple formula, where possible) is intended to assist users in assessing the effect of an argument error.

The following symbols are used in this section to describe the effect of an argument error upon the accuracy of the answer:

Symbol	Explanation
$g(x)$	The result given by the function.
$f(x)$	The correct result.
ϵ	$ <f(x) - g(x)> / f(x) $ The relative error of the result given by the function.
δ	The relative error of the argument.

$\langle \text{EPSILON} \rangle$. $|f(x) - g(x)|$ The absolute error of the result given by the function.

$\langle \text{DELTA} \rangle$ The absolute error of the argument.

The notation used for continued fractions complies with the specifications set by the United States National Bureau of Standards.²

Although it is not specifically stated in the description of each mathematical function, the algorithms in this section were programmed to conform to the following conventions governing floating-point overflow/underflow.

- Intermediate underflows and overflows are not permitted to occur. This prevents the printing of irrelevant messages.
- Those arguments for which the answer can overflow are excluded from the permitted range of the function. This rule does not apply to CABS and CDABS
- When the magnitude of the answer is less than 16 to the minus 65th power, 0 is given as the answer. If the floating-point underflow exception mask is on at the time, the underflow message will be printed.

Control of Program Exceptions in Mathematical Functions

The VS FORTRAN Version 2 mathematical functions have been implemented with careful control of error situations. A result is provided whenever the answer is within a range representable in floating-point form. To be consistent with control of exponent overflow/underflow exceptions, the following types of conditions are recognized and handled separately:

- When the magnitude of the function value is too large to be represented in floating-point form, the condition is called a terminal overflow.
- When the magnitude is too small to be represented, the condition is called a terminal underflow.
- If the function value is representable, but execution of the chosen algorithm causes an overflow or underflow in the process, this condition is called an intermediate overflow or underflow.

Function functions in the VS FORTRAN Version 2 library have been implemented to observe the following rules for these conditions:

1. Algorithms that can cause an intermediate overflow have been avoided. Therefore, an intermediate overflow should occur only rarely during the execution of a function of the library.
2. Intermediate underflows are generally detected and not allowed to cause an interrupt. In other words, spurious underflow signals are not allowed to be given. Computation of the function value is successfully carried out.
3. Terminal overflow conditions are screened out by the function. The argument is considered out of range for computation, and an error diagnostic is given.
4. Terminal underflow conditions are handled by forcing a floating-point underflow exception. This provides for the detection and treatment of underflow in the same manner used for an arithmetic statement. Terminal underflows can occur

² For more information, see Milton Abramowitz and Irene A. Stegun (editors), *Handbook of Mathematical Functions*, Applied Mathematics Series-55 (National Bureau of Standards, Washington, D.C., 1965).

| in the following function functions: ATAN2, DATAN2, ERFC, DERFC, QATAN2,
| QEXP, EXP, and DEXP.

Argument and Accuracy Information

Absolute Value Functions

ABS/IABS/DABS/QABS

CABS

Effect of an Argument Error

ε similar $x^{**2}/(a^{**2}\delta(x)) + y^{**2}/(a^{**2}\delta(y))$ where $\delta(x)$ and $\delta(y)$ are relative errors inherent in the real part and the imaginary part of the argument, respectively.

Accuracy

The accuracy of the algorithm is perfect; for all (2^{**64}) normalized and unnormalized short floating-point arguments, CABS returns the correctly rounded value of $|z|$ or an overflow message. Underflow is handled as described above.

CDABS

Effect of an Argument Error

ε similar $x^{**2}/(a^{**2}\delta(x)) + y^{**2}/(a^{**2}\delta(y))$ where $\delta(x)$ and $\delta(y)$ are relative errors inherent in the real part and the imaginary part of the argument, respectively.

Accuracy

The accuracy of the algorithm is perfect; for all (2^{**128}) normalized and unnormalized long floating-point arguments, CDABS returns the correctly rounded value of $|z|$ or an overflow message. Underflow is handled as described above.

CQABS

Effect of an Argument Error

ε similar $x^{**2}/(a^{**2}\delta(x)) + y^{**2}/(a^{**2}\delta(y))$ where $\delta(x)$ and $\delta(y)$ are relative errors inherent in the real part and the imaginary part of the argument, respectively.

Arcsine and Arccosine Functions

ASIN

Effect of an Argument Error

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle \sqrt{1 - x^{**2}}$. For small values of x , $\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle$. Toward the limits (± 1) of the range, a small $\langle \text{DELTA} \rangle$ causes a substantial error in the answer. For the arccosine, ε similar δ if the value of x is near $\pi/2$.

Accuracy

A linearly distributed sample of size 10000 produced 50 incorrectly rounded results, with 0.57 ulp the maximum observed error. One ulp is the maximum error produced by this algorithm over all arguments from -1 to 1.

ACOS

Effect of an Argument Error

ϵ *similar* $\delta \sqrt{1 - x^2}$. For small values of x , ϵ *similar* δ . Toward the limits (± 1) of the range, a small δ causes a substantial error in the answer. For the arccosine, ϵ *similar* δ if the value of x is near $\pi/2$.

Accuracy

A linearly distributed sample of size 10000 produced 95 incorrectly rounded results, with 0.64 ulp the maximum observed error. One ulp is the maximum error produced by this algorithm over all arguments from -1 to 1.

DASIN

Effect of an Argument Error

ϵ *similar* $\delta \sqrt{1 - x^2}$. For small values of x , ϵ *similar* δ . Toward the limits (± 1) of the range, a small δ causes a substantial error in the answer. For the arcsine, ϵ *similar* δ if the value of x is small.

Accuracy

A linearly distributed sample of size 10000 produced 542 incorrectly rounded results, with 0.80 ulp the maximum observed error. One ulp is the maximum error produced by this algorithm over all arguments from -1 to 1.

DACOS

Effect of an Argument Error

ϵ *similar* $\delta \sqrt{1 - x^2}$. For small values of x , ϵ *similar* δ . Toward the limits (± 1) of the range, a small δ causes a substantial error in the answer. For the arccosine, ϵ *similar* δ if the value of x is near $\pi/2$.

Accuracy

A linearly distributed sample of size 10000 produced 205 incorrectly rounded results, with 0.79 ulp the maximum observed error. One ulp is the maximum error produced by this algorithm over all arguments from -1 to 1.

QARSIN/QARCOS***Effect of an Argument Error***

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (1 - x^2)$. For a small value of x , $\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle$. Toward the limits (± 1) of the range, a small $\langle \text{DELTA} \rangle$ causes a substantial error in the answer. For the arcsin, ϵ similar δ if the value of x is small.

Arctangent Functions**ATAN*****Effect of an Argument Error***

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (1 + x^2)$. For small values of x , ϵ similar δ ; as the value of x increases, the effect of δ upon ϵ diminishes.

Accuracy

The accuracy of the algorithm is nearly perfect. For sample sizes of 10000 linearly distributed arguments in the range $0 \leq x \leq 10$, the algorithm returns the correctly rounded result 97% of the time, with a maximum observed error of 0.60 ulp. For sample sizes of 10000 logarithmically distributed arguments in the range $10^{(-5)} < x < 10^{(10)}$, the algorithm returns the correctly rounded result 98.4% of the time, with a maximum observed error of 0.59 ulp. Over the entire range of short floating-point numbers, the maximum error is under 0.66 ulp.

ATAN2***Effect of an Argument Error***

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (1 + x^2)$. For small values of x , ϵ similar δ , and as the value of x increases, the effect of ϵ upon δ diminishes.

Accuracy

The accuracy of the algorithm is nearly perfect. For sample sizes of 10000 linearly distributed arguments in the range $0 \leq y \leq 10$, $x = 1.0$, the algorithm returns the correctly rounded result 98.8% of the time, with a maximum observed error of 0.60 ulp. For sample sizes of 10000 logarithmically distributed arguments in the range $10^{(-5)} < y < 10^{(10)}$, $x = 1.0$, the algorithm returns the correctly rounded result 99.5% of the time, with a maximum observed error of 0.56 ulp. Over the entire range of short floating-point numbers, the maximum error is under one ulp.

DATAN***Effect of an Argument Error***

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (1 + x^2)$. For small values of x , ϵ similar δ , as the value of x increases, the effect of ϵ upon δ diminishes.

Accuracy

The accuracy of the algorithm is nearly perfect. For sample sizes of 10000 linearly distributed arguments in the range $0 \leq x \leq 10$, the algorithm returns the correctly rounded result 97% of the time, with a maximum observed error of 0.60 ulp. For sample sizes of 10000 logarithmically distributed arguments in the range $10^{**}(-10) < x < 10^{**}(20)$, the algorithm returns the correctly rounded result 99.7% of the time, with a maximum observed error of 0.53 ulp. Over the entire range of long floating-point numbers, the maximum error is under one ulp.

DATAN2

Effect of an Argument Error

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (1 + x^{**2})$. For small values of x , ϵ similar δ , as the value of x increases, the effect of ϵ upon δ diminishes.

Accuracy

For sample sizes of 10000 linearly distributed arguments in the range $0 \leq y \leq 10$, $x = 1.0$, the algorithm returns the correctly rounded result 94% of the time, with a maximum observed error of 0.57 ulp. For sample sizes of 10000 logarithmically distributed arguments in the range $10^{**}(-10) < y < 10^{**}(20)$, $x = 1.0$ the algorithm returns the correctly rounded result 87.5% of the time, with a maximum observed error of 0.75 ulp. Over the entire range of short floating-point numbers, the maximum error is under one ulp.

QATAN/QATAN2

Effect of an Argument Error

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (1 + x^{**2})$. For a small value of x , ϵ similar δ , and as the value of x increases, the effect of δ upon ϵ diminishes.

Error Function Functions

ERF/ERFC

Effect of an Argument Error

E similar $e^{**}(-x^{**2}) * \langle \text{DELTA} \rangle$. For the error function, as the magnitude of the argument exceeds 1, the effect of an argument error upon the final accuracy diminishes rapidly. For small values of x , ϵ similar δ . For the complemented error function, if the value of x is greater than 1, $\text{erfc}(x)$ similar $e^{**}(-x^{**2}) > 2x$. Therefore, ϵ similar $2(x^{**2}) * \delta$. If the value of x is negative or less than 1, then ϵ similar $e^{**}(-x^{**2}) * \langle \text{DELTA} \rangle$.

DERF/DERFC***Effect of an Argument Error***

E similar $e^{**}(-x^{**2}) * <DELTA>$. For the error function, as the magnitude of the argument exceeds 1, the effect of an argument error upon the final accuracy diminishes rapidly. For small values of x , ϵ similar δ . For the complemented error function, if the value of x is greater than 1, $erfc(x)$ similar $(e^{**}(-x^{**2}))/2x$. Therefore, ϵ similar $(2x^{**2} * \delta)$. If the value of x is negative or less than 1, then ϵ similar $e^{**}(-x^{**2}) * <DELTA>$.

QERF/QERFC***Effect of an Argument Error***

$<EPSILON>$ similar $e^{**}(-x^{**2}) * <DELTA>$. For the error function, as the magnitude of the argument exceeds 1, the effect of an argument error upon the final accuracy diminishes rapidly. For small values of x , ϵ similar δ . For the complemented error function, if the value of x is greater than 1, then $erfc(x)$ similar $(e^{**}(-x^{**2}))/2x$. Therefore, ϵ similar $(2x^{**2} * \delta)$. If the value of x is negative or less than 1, then ϵ similar $e^{**}(-x^{**2}) * <DELTA>$.

Exponential Functions**EXP*****Effect of an Argument Error***

ϵ similar $<DELTA>$. If the magnitude of x is large, even the round-off error of the argument causes a substantial relative error in the answer because $<DELTA> = \delta * x$.

Accuracy

The accuracy of the algorithm is perfect; that is, for every argument x , normalized or unnormalized, $EXP(x)$ returns the correctly rounded result. An exhaustive test has shown that the correction made in Step 5 has sufficient accuracy to compute the correctly rounded result for all x .

DEXP***Effect of an Argument Error***

ϵ similar $<DELTA>$. If the magnitude of x is large, even the round-off error of the argument causes a substantial relative error in the answer because $<DELTA> = \delta * x$.

Accuracy

The accuracy of the algorithm is nearly perfect. For a linearly distributed sample of 20000 random arguments in the range -174.673 to 174.673, the algorithm returned the correctly rounded result 99.8% of the time, with a maximum observed error of 0.53 ulp. Over the entire range of long floating-point arguments, the maximum error is under one ulp.

CEXP/CDEXP

Effect of an Argument Error

The effect of an argument error depends upon the accuracy of the individual parts of the argument. If $e^{x+iy} = R * e^{iH}$, then $H = y$ and $\varepsilon(R) \text{ similar } \langle \text{DELTA} \rangle(x)$.

QEXP

Effect of an Argument Error

$\varepsilon \text{ similar } \langle \text{DELTA} \rangle$. If the magnitude of x is large, even the round-off error of the argument causes a substantial relative error in the answer because $\langle \text{DELTA} \rangle = \delta * x$.

CQEXP

Effect of an Argument Error

The effect of an argument error depends upon the accuracy of the individual parts of the argument. If $e^{x+iy} = R * e^{iH}$, then $H = y$ and $\varepsilon(R) \text{ similar } \langle \text{DELTA} \rangle(x)$.

Gamma and Log Gamma Functions

GAMMA/ALGAMA

Effect of an Argument Error

$\varepsilon \text{ similar } \langle \text{psi} \rangle(x) * \langle \text{DELTA} \rangle$ for gamma, and $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{psi} \rangle(x) * \langle \text{DELTA} \rangle$ for log-gamma, where $\langle \text{psi} \rangle$ is the digamma function. If $1/2 < x < 3$, then $-2 < \langle \text{psi} \rangle(x) < 1$. Therefore, $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{DELTA} \rangle$ for log-gamma. However, because $x = 1$ and $x = 2$ are zeros of the log-gamma function, even a small δ can cause a substantial ε in this range.

If the value of x is large, then $\langle \text{psi} \rangle(x) \text{ similar } \text{logsube}(x)$. Therefore, for gamma, $\varepsilon \text{ similar } \delta * x * \text{logsube}(x)$. In this case, even the round-off error of the argument contributes greatly to the relative error of the answer. For log-gamma with large values of x , $\varepsilon \text{ similar } \delta$.

DGAMMA/DLGAMA

Effect of an Argument Error

$\varepsilon \text{ similar } \langle \text{psi} \rangle(x) * \langle \text{DELTA} \rangle$ for gamma, and $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{psi} \rangle(x) * \langle \text{DELTA} \rangle$ for log-gamma, where $\langle \text{psi} \rangle$ is the digamma function. If $1/2 < x < 3$, then $-2 < \langle \text{psi} \rangle(x) < 1$. Therefore $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{DELTA} \rangle$ for log-gamma. However, because $x = 1$ and $x = 2$ are zeros of the log-gamma function, even a small δ can cause a substantial ε in this range.

If the value of x is large, then $\langle \text{psi} \rangle(x) \text{ similar } \text{logsube}(x)$. Therefore, for gamma, $\varepsilon \text{ similar } \delta * x * \text{logsube}(x)$. In this case, even the round-off error of the argument contributes greatly to the relative error of the answer. For log-gamma with large values of x , $\varepsilon \text{ similar } \delta$.

Hyperbolic Sine and Cosine Functions

SINH/COSH

Effect of an Argument Error

For the hyperbolic sine, $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{DELTA} \rangle * \cosh(x)$ and $\varepsilon \text{ similar } \langle \text{DELTA} \rangle * \coth(x)$. For the hyperbolic cosine, $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{DELTA} \rangle * \sinh(x)$ and $\varepsilon \text{ similar } \delta * \tanh(x)$. Specifically, for the cosine, $\varepsilon \text{ similar } \langle \text{DELTA} \rangle$ over the entire range; for the sine, $\varepsilon \text{ similar } \delta$ for small values of x .

Accuracy

The accuracy of the algorithm is perfect; that is, for every argument x , normalized or unnormalized, $\text{SINH/COSH}(x)$ returns the correctly rounded result. An exhaustive test has shown that this algorithm has sufficient accuracy to compute the correctly rounded result for all x .

DSINH/DCOSH

Effect of an Argument Error

For the hyperbolic sine, $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{DELTA} \rangle * \cosh(x)$ and $\varepsilon \text{ similar } \langle \text{DELTA} \rangle * \coth(x)$. For the hyperbolic cosine, $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{DELTA} \rangle * \sinh(x)$ and $\varepsilon \text{ similar } \delta * \tanh(x)$. Specifically, for the cosine, $\varepsilon \text{ similar } \langle \text{DELTA} \rangle$ over the entire range; for the sine, $\varepsilon \text{ similar } \delta$ for small values of x .

QSINH/QCOSH

Effect of an Argument Error

For the hyperbolic sine, $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{DELTA} \rangle * \cosh(x)$ and $\varepsilon \text{ similar } \langle \text{DELTA} \rangle * \coth(x)$. For the hyperbolic cosine, $\langle \text{EPSILON} \rangle \text{ similar } \langle \text{DELTA} \rangle * \sinh(x)$. and $\varepsilon \text{ similar } \delta * \tanh(x)$. In other words, for cosine, $\varepsilon \text{ similar } \langle \text{DELTA} \rangle$ over the entire range; for the sine, $\varepsilon \text{ similar } \delta$ for small values of x .

Hyperbolic Tangent Functions

TANH

Effect of an Argument Error

$\langle \text{EPSILON} \rangle \text{ similar } (1 - \tanh^2(x)) * \langle \text{DELTA} \rangle$, and $\varepsilon \text{ similar } (2 * \langle \text{DELTA} \rangle) / (\sinh(2x))$. For small values of x , $\varepsilon \text{ similar } \delta$, and, as the value of x increases, the effect of δ upon ε diminishes.

Accuracy

The accuracy of the algorithm is perfect; that is, for every argument x , normalized or unnormalized, $\text{TANH}(x)$ returns the correctly rounded result. An exhaustive test has shown that this algorithm has sufficient accuracy to compute the correctly rounded result for all x .

DTANH

Effect of an Argument Error

<EPSILON> similar $(1 - \tanh^2(x))\langle\text{DELTA}\rangle$, and ε similar $(2\langle\text{DELTA}\rangle) / (\sinh(2x))$. For small values of x , ε similar δ . As the value of x increases, the effect of δ upon ε diminishes.

QTANH

Effect of an Argument Error

<EPSILON> similar $(1 - \tanh^2(x))\langle\text{DELTA}\rangle$, and ε similar $(2\langle\text{DELTA}\rangle) / (\sinh(2x))$. For small values of x , ε similar δ . As the value of x increases, the effect of δ upon ε diminishes.

Logarithmic Functions (Common and Natural)

LOG/LOG10

ALOG/ALOG10

Accuracy

In a test run with 10000 random logarithmically distributed x values in the interval $1.92874989 \times 10^{(-22)} < x < 5.18470152 \times 10^{(21)}$, no errors were found; that is, the results were all equal to correctly rounded exact values. In another test run with 10000 equally spaced x values in the interval $0.93 < x < 1.07$, 99.7% of the results were correct and the remainder had a maximum error of 0.54 ulp. A third test with 10000 equally spaced x values in a smaller interval, $0.995 < x < 1.004$, showed 98.2% of the results to be correct, while the remainder had a maximum error of only 0.56 ulp. Similar tests, with various distributions of several hundreds of thousands of random numbers in the legal range, showed similar results with no error greater than 1 ulp.

DLOG/DLOG10

Effect of an Argument Error

<EPSILON> similar δ . Therefore, if the value of the argument is close to 1, the relative error can be very large because the value of the function is very small.

Accuracy

In tests with over 30000 logarithmically distributed random numbers covering the entire range of possible arguments, the double precision routines were found to give exact results for about 96.5% of the trials. The largest error found, in extensive testing around ulp boundaries, was 0.97 ulp.

QLOG/QLOG10

Effect of an Argument Error

<EPSILON> similar δ . Therefore, if the value of the argument is close to 1, the relative error can be very large, because the value of the function is very small.

CLOG/CDLOG

Effect of an Argument Error

The effect of an argument error depends upon the accuracy of the individual parts of the argument. If $x + iy = r * e^{iH}$ and $\logsube(x + iy) = a + ib$, then $H = b$ and $E(a) = \delta(r)$.

CQLOG

Effect of an Argument Error

<EPSILON> similar δ . Therefore, if the argument is close to 1, the relative error can be very large, because the value of the function is very small.

Sine and Cosine Functions

SIN

Effect of an Argument Error

<EPSILON> similar <DELTA>. As the value of the argument increases, <DELTA> increases. Because the function value diminishes periodically, no consistent relative error control can be maintained outside of the principal range, $-\pi/2 \leq x \leq +\pi/2$.

Accuracy

The accuracy of the SIN algorithm is nearly perfect. For a sample size of 10000 linearly distributed arguments in the range $-\pi < x < \pi$, the SIN algorithm returned the correctly rounded result 98.2% of the time with a maximum observed error of 0.55 ulp. For a sample size of 10000 logarithmically distributed arguments in the range $2^{*(-18)}\pi < x < 2^{*18}\pi$, the SIN algorithm returned the correctly rounded result 98.8% of the time, with a maximum observed error of 0.55 ulp. Over the entire range of acceptable arguments ($|x| < 2^{*18}\pi$), the maximum error is under 0.70 ulp for this algorithm.

COS

Effect of an Argument Error

<EPSILON> similar <DELTA>. As the value of the argument increases, <DELTA> increases. Because the function value diminishes periodically, no consistent relative error control can be maintained outside of the principal range, $-\pi/2 \leq x \leq +\pi/2$.

Accuracy

The accuracy of the COS algorithm is nearly perfect. For a sample size of 10000 linearly distributed arguments in the range $-\pi < x < \pi$, the COS algorithm returned the correctly rounded result 98.4% of the time, with a maximum observed error of 0.55 ulp. For a sample size of 10000 logarithmically distributed arguments in the range $2^{*(-18)}\pi < x < 2^{*18}\pi$, the COS algorithm returned the correctly rounded result 99.0% of the time, with a maximum observed error of 0.54 ulp. Over the entire range of acceptable arguments ($|x| < 2^{*18}\pi$), the maximum error is under 0.70 ulp for this algorithm.

DSIN

Effect of an Argument Error

<EPSILON> similar <DELTA>. As the value of the argument increases, *<DELTA>* increases. Because the function value diminishes periodically, no consistent relative error control can be maintained outside of the principal range, $-\pi/2 \leq x \leq +\pi/2$.

Accuracy

The accuracy of the DSIN algorithm is nearly perfect. For a sample size of 10000 linearly distributed arguments in the range $-\pi < x < \pi$, the DSIN algorithm returned the correctly rounded result 96.7% of the time, with a maximum observed error of 0.57 ulp. For a sample size of 10000 logarithmically distributed arguments in the range $2^{**}(-50)\pi < x < 2^{**}(50)\pi$, the DSIN algorithm returned the correctly rounded result 97.4% of the time, with a maximum observed error of 0.57 ulp. Over the entire range of acceptable arguments ($|x| < 2^{**}(50)\pi$), the maximum error is under one ulp for this algorithm.

DCOS

Effect of an Argument Error

<EPSILON> similar <DELTA> . As the value of the argument increases, *<DELTA>* increases. Because the function value diminishes periodically, no consistent relative error control can be maintained outside of the principal range, $-\pi/2 \leq x \leq +\pi/2$.

Accuracy

The accuracy of the DCOS algorithm is nearly perfect. For a sample size of 10000 linearly distributed arguments in the range $-\pi < x < \pi$, the DCOS algorithm returned the correctly rounded result 96.3% of the time, with a maximum observed error of 0.57 ulp. For a sample size of 10000 logarithmically distributed arguments in the range $2^{**}(-50)\pi < x < 2^{**}(50)\pi$, the DCOS algorithm returned the correctly rounded result 96.6% of the time, with a maximum observed error of 0.56 ulp. Over the entire range of acceptable arguments ($|x| < 2^{**}(50)\pi$), the maximum error is under one ulp for this algorithm.

QSIN/QCOS

Effect of an Argument Error

<EPSILON> similar <DELTA>. As the value of x increases, *<DELTA>* increases. Because the function value diminishes periodically, no consistent relative error control can be normally maintained outside the principal range $-\pi/2 \leq x \leq +\pi/2$.

CSIN/CCOS

Effect of an Argument Error

The effect of an argument error upon the accuracy of the answer depends on the effect of an argument error in the SIN/COS, EXP, and SINH/COSH functions.

CDSIN/CDCOS***Effect of an Argument Error***

The effect of an argument error upon the accuracy of the answer depends on the effect of an argument error in the DSIN/DCOS, DEXP, and DSINH/DCOSH functions.

CQSIN/CQCOS***Effect of an Argument Error***

The effect of an argument error upon the accuracy of the answer depends on the effect of an argument error in the DSIN/DCOS, DEXP, and DSINH/DCOSH functions.

Square Root Functions**SQRT*****Effect of an Argument Error***

ε similar $\delta/2$.

Accuracy

The accuracy of the algorithm is perfect; for all $(15 * 2^{27})$ nonnegative normalized short floating-point numbers x , SQRT returns the correctly rounded value z of \sqrt{x} . The accuracy of the algorithm is additionally perfect for all nonnegative unnormalized short floating-point numbers, including those with -64 exponent.

DSQRT***Effect of an Argument Error***

ε similar $\delta/2$.

Accuracy

The accuracy of the algorithm is perfect; for all $(15 * 2^{59})$ nonnegative normalized long floating-point numbers x , DSQRT returns the correctly rounded value z of \sqrt{x} . The accuracy of the algorithm is additionally perfect for all nonnegative unnormalized long floating-point numbers, including those with -64 exponent.

QSQRT***Effect of an Argument Error***

ε similar $\delta/2$

CSQRT/CDSQRT

Effect of an Argument Error

The effect of an argument error depends upon the accuracy of the individual parts of the argument. If $x + iy = r * e^{**}(ih)$ and $\sqrt{x + iy} = R * e^{**}(iH)$, then $\varepsilon(R)$ similar $(\delta/2)^*(r)$, and $\varepsilon(H)$ similar $\delta(h)$.

CQSQRT

Effect of an Argument Error

Using polar coordinates, write $x + iy = r * e^{**}(ih)$ and $\sqrt{x + iy} = R * e^{**}(iH)$. Then $\varepsilon(R)$ similar $(\delta(r))/2$, and $\varepsilon(H)$ similar $\delta(h)$.

Tangent and Cotangent Functions

TAN

Effect of an Argument Error

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (\cos^{**2}(x))$, and ε similar $2/\sin(2x)$ for $\tan(x)$. Therefore, near the singularities $x = (k + 1/2)\pi$, where k is an integer, no error control can be maintained.

Accuracy

The accuracy of the algorithm is nearly perfect. For a sample size of 10000 linearly distributed arguments in the range $-\pi/2 < x < \pi/2$, the algorithm returned the correctly rounded result 97% of the time, with a maximum observed error of 0.86 ulp. For a sample size of 10000 logarithmically distributed arguments in the range $2^{**}(-18)\pi < x < 2^{**}(18)\pi$, the algorithm returned the correctly rounded result 97.7% of the time, with a maximum observed error of 0.78 ulp. Over the entire range of acceptable arguments ($|x| < 2^{**}(18)\pi$), the maximum error is less than 0.91 ulp.

COTAN

Effect of an Argument Error

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (\sin^{**2}(x))$, and ε similar $(2\langle \text{DELTA} \rangle) / (|\sin(2x)|)$ for $\tan(x)$. Therefore, near the singularities $x = k\pi$, where k is an integer, no error control can be maintained.

Accuracy

The accuracy of the algorithm is nearly perfect. For a sample size of 10000 linearly distributed arguments in the range $-\pi/2 < x < \pi/2$, the algorithm returned the correctly rounded result 96.7% of the time, with a maximum observed error of 0.83 ulp. For a sample size of 10000 logarithmically distributed arguments in the range $2^{**}(-18)\pi < x < 2^{**}(18)\pi$, the algorithm returned the correctly rounded result 97.5% of the time, with a maximum observed error of 0.85 ulp. Over the entire range of acceptable arguments ($|x| < 2^{**}(18)\pi$), the maximum error is under 0.99 ulp.

DTAN

Effect of an Argument Error

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (\cos^2(x))$, and ε similar $2/\sin(2x)$. Therefore, near the singularities of $x = (k + 1/2)\pi$, where k is an integer, no error control can be maintained.

Accuracy

The accuracy of the algorithm is nearly perfect. For sample sizes of 10000 linearly distributed arguments in the range $-\pi/2 < x < \pi/2$, the algorithm returns the correctly rounded result 95% of the time, with a maximum observed error of 0.89 ulp. For sample sizes of 10000 logarithmically distributed arguments in the range $2^{**}(-50) \pi < x < 2^{**}(50) \pi$, the algorithm returns the correctly rounded result 97% of the time, with a maximum observed error of 0.83 ulp. Over the entire range of acceptable arguments ($|x| < 2^{**}(50) \pi$), the maximum error is less than one ulp.

DCOTAN

Effect of an Argument Error

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / \sin^2(x)$, and ε similar $2\langle \text{DELTA} \rangle / (|\sin(2x)|)$. Therefore, near the singularities $x = k\pi$, where k is an integer, no error control can be maintained.

Accuracy

The accuracy of the algorithm is nearly perfect. For a sample size of 10000 linearly distributed arguments in the range $-\pi/2 < x < \pi/2$, the algorithm returned the correctly rounded result 95% of the time, with a maximum observed error of 0.90 ulp. For a sample size of 10000 logarithmically distributed arguments in the range $2^{**}(-50) \pi < x < 2^{**}(50) \pi$, the algorithm returned the correctly rounded result 97% of the time, with a maximum observed error of 0.83 ulp. Over the entire range of acceptable arguments ($|x| < 2^{**}(50) \pi$), the maximum error is under one ulp.

QTAN/QCOTAN

Effect of an Argument Error

$\langle \text{EPSILON} \rangle$ similar $\langle \text{DELTA} \rangle / (\cos^2(x))$, and ε similar $(2\langle \text{DELTA} \rangle) / (\sin(2x))$ for $\tan(x)$. Therefore, near the singularities $x = (k + 1/2)\pi$, where k stands for integers, no error control can be maintained. This is also true of $\cot(x)$ for $x \text{ near } k\pi$, where k is an integer.

Part 2. Library Reference

Chapter 7. 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 either explicitly or implicitly called.

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 6, "Intrinsic Functions" on page 280.

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 37 and Figure 38 on page 313. 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 39 on page 314.

For routines that involve exponentiation, the action taken within a routine depends upon the types of the base and exponent used. Figure 40 on page 314 through Figure 43 on page 315 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 about calling the routines from the assembler language is given in Appendix B, "Assembler Language Information" on page 412.

Figure 37 (Page 1 of 4). Implicitly Called Mathematical Routines

General Function	Entry Name(1)	Implicit Function Reference(2)	Arg. Number	Argument Type(3)	Function Value Type(3)	Error Code
Multiply and divide complex numbers	CMPY#	y = zsub1 * zsub2	2	COMPLEX*8	COMPLEX*8	
	CDVD#	y = zsub1/zsub2	2	COMPLEX*8	COMPLEX*8	
	CDMPY#	y = zsub1 * zsub2	2	COMPLEX*16	COMPLEX*16	

Figure 37 (Page 2 of 4). Implicitly Called Mathematical Routines

General Function	Entry Name(1)	Implicit Function Reference(2)	Arg. Number	Argument Type(3)	Function Value Type(3)	Error Code
	CDDVD#	$y = \text{zsub1}/\text{zsub2}$	2	COMPLEX*16	COMPLEX*16	
	CQMPY#	$y = \text{zsub1} * \text{zsub2}$	2	COMPLEX*32	COMPLEX*32	
	CQDVD#	$y = \text{zsub1}/\text{zsub2}$	2	COMPLEX*32	COMPLEX*32	
Compare complex numbers	CXMPR# Note 4	$y = \text{zsub1 compop zsub2}$ Note 5	2	COMPLEX (of all lengths)	LOGICAL*4	
Raise an integer to an integer power	FIXPI#	$y = i ** j$	2	$i = \text{INTEGER*1 or INTEGER*2 or INTEGER*4 or BYTE or UNSIGNED*1}$ $j = \text{INTEGER*1 or INTEGER*2 or INTEGER*4 or BYTE or UNSIGNED*1}$	Same as i	241
	F8XPI#	$y = i ** j$	2	$i = \text{INTEGER*8}$ $j = \text{INTEGER*1 or INTEGER*2 or INTEGER*4 or BYTE or UNSIGNED*1}$	INTEGER*8	302
	FIXP8#	$y = i ** j$	2	$i = \text{INTEGER*1 or INTEGER*2 or INTEGER*4 or BYTE or UNSIGNED*1}$ $j = \text{INTEGER*8}$	Same as i	303
	F8XP8#	$y = i ** j$	2	$i = \text{INTEGER*8}$ $j = \text{INTEGER*8}$	INTEGER*8	304
Raise a real number to an integer power	FRXPI# Note 7	$y = a ** j$	2	$a = \text{REAL*4}$ $j = \text{INTEGER*1 or INTEGER*2 or INTEGER*4 or BYTE or UNSIGNED*1}$	REAL*4	242
	FRXP8#	$y = a ** j$	2	$a = \text{REAL*4}$ $j = \text{INTEGER*8}$	REAL*8	305
	FDXPI# Note 7	$y = a ** j$	2	$a = \text{REAL*8}$ $j = \text{INTEGER*1 or INTEGER*2 or INTEGER*4 or BYTE or UNSIGNED*1}$	REAL*8	243
	FDXP8#	$y = a ** j$	2	$a = \text{REAL*8}$ $j = \text{INTEGER*8}$	REAL*8	306
	FQXPI# Note 7	$y = a ** j$	2	$a = \text{REAL*16}$ $j = \text{INTEGER*1 or INTEGER*2 or INTEGER*4 or BYTE or UNSIGNED*1}$	REAL*16	248
	FQXP8#	$y = a ** j$	2	$a = \text{REAL*16}$ $j = \text{INTEGER*8}$	REAL*16	307

Figure 37 (Page 3 of 4). Implicitly Called Mathematical Routines

General Function	Entry Name(1)	Implicit Function Reference(2)	Arg. Number	Argument Type(3)	Function Value Type(3)	Error Code
Raise a real number to a real power	FRXPR# Note 6	$y = a^{**} b$	2	$a = \text{REAL}^*4$ $b = \text{REAL}^*4$	REAL^*4	118, 244
	FDXPD# Note 6	$y = a^{**} b$	2	$a = \text{REAL}^*8$ $b = \text{REAL}^*8$	REAL^*8	119, 245
	FQXPQ#	$y = a^{**} b$	2	$a = \text{REAL}^*16$ $b = \text{REAL}^*16$	REAL^*16	249 250
Raise 2 to a real power	FQXP2#	$y = 2^{**} b$	1	$b = \text{REAL}^*16$	REAL^*16	260
Raise a complex number to an integer power	FCXPI#	$y = z^{**} j$	2	$z = \text{COMPLEX}^*8$ $j = \text{INTEGER}^*1$ or INTEGER^*2 or INTEGER^*4 BYTE or UNSIGNED*1	COMPLEX^*8	246
	FCXP8#	$y = z^{**} j$	2	$z = \text{COMPLEX}^*8$ $j = \text{INTEGER}^*8$	COMPLEX^*8	308
	FCDXI#	$y = z^{**} j$	2	$z = \text{COMPLEX}^*16$ $j = \text{INTEGER}^*1$ or INTEGER^*2 or INTEGER^*4 BYTE or UNSIGNED*1	COMPLEX^*16	247
	FCDX8#	$y = z^{**} j$	2	$z = \text{COMPLEX}^*16$ $j = \text{INTEGER}^*8$	COMPLEX^*16	309
	FCQXI#	$y = z^{**} j$	2	$z = \text{COMPLEX}^*32$ $j = \text{INTEGER}^*1$ or INTEGER^*2 or INTEGER^*4 BYTE or UNSIGNED*1	COMPLEX^*32	270
	FCQX8#	$y = z^{**} j$	2	$z = \text{COMPLEX}^*32$ $j = \text{INTEGER}^*8$	COMPLEX^*32	310
Raise a complex number to a complex power	FCXPC#	$y = a^{**} b$	2	$a = \text{COMPLEX}^*8$ $b = \text{COMPLEX}^*8$	COMPLEX^*8	246
	FCD CD#	$y = a^{**} b$	2	$a = \text{COMPLEX}^*16$ $b = \text{COMPLEX}^*16$	COMPLEX^*16	247

Figure 37 (Page 4 of 4). Implicitly Called Mathematical Routines

General Function	Entry Name(1)	Implicit Function Reference(2)	Arg. Number	Argument Type(3)	Function Value Type(3)	Error Code
	FCQCQ#	y = a ** b	2	a = COMPLEX*32 b = COMPLEX*32	COMPLEX*32	270

Notes:

- (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, REAL*16, and COMPLEX*16 arguments correspond to real, double precision, extended precision, and double complex arguments, respectively, in VS FORTRAN Version 2.
The effect of argument error for some of these routines can be found in "Error Conditions for Implicitly Called Functions" on page 315.
Function Value Types shown as INTEGER*1, INTEGER*2, UNSIGNED*1, and BYTE are actually returned from the function as INTEGER*4.
- (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. or == or /= or <>).
- (6) Available also in the alternative mathematical library.
- (7) When j is negative, overflow may occur (AFB207I) rather than underflow (AFB208I).

Figure 38. Implicitly Called Character Routines

Entry Name	Implicit Function Reference	Arg(s) Number	Argument(s) Type	Function Value Type	Error Code
CCMPR# Note 2	y = xsub1 compop xsub2 Note 1	2	CHARACTER	LOGICAL	193
CMOVE# Note 2	y = x	2	CHARACTER	CHARACTER	195 196 197
CNCAT#	y = xsub1//xsub2 ...xsubn//xsubn	>= 2	CHARACTER	CHARACTER	199

Notes:

- (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 and later releases, 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.

Figure 39. Implicitly Called Service Routines

Entry Name	Function	Arguments	Error Code
DSPAN# DSPN2# DSPN4# DSPN5#	Calculate dimension factors and span of adjustable dimension array.	Array description	187
DYCMN#	For dynamic common block, obtain storage and relocate adcons.	Common block and adcon information	156 158
DYCMNS# ECCMN#	Manage common blocks. For static common blocks, record their use and relocate adcons, if necessary. For dynamic and extended common blocks, obtain storage and relocate adcons.		

Figure 40. Exponentiation with Integer Base and Exponent

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 41. Exponentiation with Real Base and Integer 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 42 (Page 1 of 2). Exponentiation with Real Base and 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

Figure 42 (Page 2 of 2). Exponentiation with Real Base and Exponent

Base (A)	Exponent (B) B > 0	Exponent (B) B = 0	Exponent (B) B < 0
A < 0	Error message 118 or 119	Function value = 1	Error message 118 or 119

Figure 43. Exponentiation with Complex Base and Integer 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 + 0i	Compute the function value
P > 0 and Q = 0	Compute the function value	Function value = 1 + 0i	Compute the function value
P > 0 and Q < 0	Compute the function value	Function value = 1 + 0i	Compute the function value
P = 0 and Q > 0	Compute the function value	Function value = 1 + 0i	Compute the function value
P = 0 and Q = 0	Function value 0 + 0i	Error message 246 or 247	Error message 246 or 247
P = 0 and Q < 0	Compute the function value	Function value = 1 + 0i	Compute the function value
P < 0 and Q > 0	Compute the function value	Function value = 1 + 0i	Compute the function value
P < 0 and Q = 0	Compute the function value	Function value = 1 + 0i	Compute the function value
P < 0 and Q < 0	Compute the function value	Function value = 1 + 0i	Compute the function value

Error Conditions for Implicitly Called Functions

The entry point names of the following implicitly called functions are generated by the compiler. See Figure 37 on page 310 for their definitions.

Complex Divide and Multiply Functions

CDVD#/CMPY# (COMPLEX*8 Arguments)

CDDVD#/CDMPY# (COMPLEX*16 Arguments)

Error Conditions

Partial underflows can occur in preparing the answer.

CQDPY#/CQMVD# (COMPLEX*32 Arguments)

Effect of an Argument Error

In terms of complex vector relative errors, $\varepsilon \sim \delta x + \delta y$ where δx is the relative error of the first operand and δy is the relative error of the second operand.

Exponentiation of a Real Base to an Integer Power

FRXPI# (REAL*4 Argument)

FDXPI# (REAL*8 Argument)

FQXPI# (REAL*16 Argument)

Effect of an Argument Error

$$\varepsilon \sim J\delta$$

Exponentiation of a Base Argument to a Real Power

FQXP2# (REAL*16 Argument)

This subprogram uses the same algorithm as the QEXP subprogram.

Exponentiation of a Real Base to a Real Power

FRXPR# Single Precision Exponentiation ($X^{**}Y$)

FDXPD# (REAL*8 Arguments)

Effect of an Argument Error

$$(x(1 + \delta_{\text{sub1}})) y(1 + \delta_{\text{sub2}}) \equiv x^{**}y(1 + \delta_{\text{sub2}}y * \log(x) + y\delta_{\text{sub1}}).$$

Note that, if the answer does not overflow,

$$|y * \log(x)| < 175$$

On the other hand, y can be very large without causing an overflow of $x^{**}y$ if $\log(x)$ is very small. Thus, if $x \approx 1$ and if y is very large, then the effect of the perturbation δ_{sub1} of x shows very heavily in the relative error of the answer.

Accuracy

In test runs of many tens of thousands of arguments, exact rounded results were obtained for about 96.4% of the cases and no error of more than 1 ulp was found. This agrees with an analysis of the errors in the algorithm.

FQXPQ# (REAL*16 Arguments)

Effect of an Argument Error

$\varepsilon \sim y\delta_{\text{subx}} + y\log(x)\delta_{\text{suby}}$. The factor $y\log(x)$ is limited to 180 in magnitude. Beyond this, the result will overflow. However, the other factor y can be very large if x is close to 1. This is because $\log(x)$ will then be very small.

Complex Exponentiation

Exponentiation of a Complex Base to an Integer Power

FCXPI# (COMPLEX*8 Arguments)

FCDXI# (COMPLEX*16 Arguments)

FCQXI# (COMPLEX*32 Arguments)

Effect of an Argument Error

$|\varepsilon| \sim J|\delta|$, where δ is the complex relative error of the base and ε is the complex relative error of the result.

Exponentiation of a Complex Base to a Complex Power

FCXPC# (COMPLEX*8 Arguments)

FCDCD# (COMPLEX*16 Arguments)

FCQCQ# (COMPLEX*32 Arguments)

Effect of an Argument Error

If $z_{sub1} = x_{sub1} + i y_{sub1}$, and $z_{sub2} = x_{sub2} + i y_{sub2}$ then
 $z_{sub1} ** z_{sub2} = \exp(a) * (\cos(b) + i \sin(b))$, where
 $a = x_{sub2} * \log|x_{sub1} + i y_{sub1}| - y_{sub2} * \arctan(y_{sub1}/x_{sub1})$
and $b = y_{sub2} * \log|x_{sub2} + i y_{sub2}| + x_{sub2} * \arctan(y_{sub1}/x_{sub1})$.

The function $z_{sub1} ** z_{sub2}$ is calculated, using the appropriate FORTRAN routines for sin, cos, exp, log, and arctan of the required precision. Therefore, the effect of an argument error upon the accuracy of the result depends upon its effect in those functions.

Alternative Mathematical Library Subroutines

For a small subset of the standard mathematical routines, alternative routines are available that provide results compatible with VS FORTRAN Version 1. (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, MOD, AMOD, DMOD, QMOD, and for the implicitly called functions FDXPD# and FRXPR#. These alternative routines are also documented in the following manuals and describe how these routines can be installed for your use.

- *VS FORTRAN Version 2 Installation and Customization for CMS*
- *VS FORTRAN Version 2 Installation and Customization for MVS*

Figure 44 shows which libraries contain the various scalar mathematical routines for each version.

Figure 44. Libraries Containing Mathematical Routines

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

Chapter 8. Service Subroutines

The VS FORTRAN Version 2 library provides subroutines for utility programming services. The subroutines are called by the appropriate entry names in a CALL statement. VS FORTRAN Version 2 utility subroutines and functions are described in this chapter. For information on additional routines to control parallel programs, see Chapter 9, "Routines for Use with Parallel Programs" on page 344.

Mathematical Exception Subroutines

DVCHK	Tests for divide-check exception
OVERFL	Tests for exponent overflow or underflow
XUFLOW	Allows or suppresses a program interrupt caused by exponent 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

Termination Subroutines

EXIT	Ends processing of the program
SYSABN/SYSABD	Specifies abnormal termination of your job, with or without an accompanying storage dump

File Service Subroutines

FILEINF	Sets up file characteristics that will be used by an OPEN or an INQUIRE statement
UNTNOFD/UNTANY	Identifies Fortran unit numbers that are available
IGNFHU/IGNFHDD	Identifies the Fortran unit numbers and ddnames for which file history is to be ignored

Date/Time Subroutines

CLOCK/CLOCKX	Lets you determine the elapsed time of a program or a portion of a program.
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

Other Service Subroutines

ASSIGNM	Moves a character string containing double-byte data to a character variable, substring or array element, preserving balanced shift codes
ARGSTR	Retrieves command-line user parameters.
MVBITS	Assigns a bit subfield of an integer value to a bit subfield of another integer value.

For information on data-in-virtual subroutines, see Chapter 10, “Data-in-Virtual Subroutines” on page 361.

For information on extended error-handling subroutines, see Chapter 11, “Extended Error-Handling Topics” on page 382.

For information on multitasking facility subroutines, see Chapter 12, “Multitasking Facility (MTF) Subroutines” on page 401.

Mathematical Exception Routines

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 returned for k have the following meanings:

Value	Meaning
1	The divide-check indicator is <i>on</i> .
2	The divide-check indicator is <i>off</i> .

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).

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 of length 4 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.

Parallel Considerations:

For a parallel program, the XUFLOW subroutine changes the exponent underflow mask in the parallel thread that is executing.

When a subroutine is scheduled to execute in an originated task, the exponent underflow mask is set according to the XUFLOW or NOXUFLOW execution time option.

Parallel threads that execute within the same parallel task inherit the exponent underflow mask setting of the parallel thread they were dispatched from. Other parallel threads are not affected by the changes made to the exponent underflow mask unless they are combined for execution as a group.

For example, a group of iterations of a parallel loop statement may be executing as a parallel thread. If the XUFLOW subroutine is called during an iteration of the loop, the exponent underflow mask is only set for the group of iterations. Other groups of iterations might not be affected.

A nested parallel loop would inherit the exponent underflow mask setting of the iteration or group of iterations it was dispatched in.

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} (a1,b1,k1,a2,b2,k2,...)
```

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
12	UNSIGNED*1
13	INTEGER*1
14	LOGICAL*2
15	INTEGER*8
16	LOGICAL*8

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.

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 B is a character variable and TABLE is a character array of 20 elements. The following call to the storage dump routine could be used to dump TABLE and B in character format, and stop the program after the dump is taken:

```
CALL CDUMP(TABLE(1), TABLE(20), B, B)
```

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. Variables under CMS and MVS 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, saved variables in their Fortran-defined data representation.
- All variables contained in a static common area (blank or named) in their Fortran-defined data representation.
- All variables in CMS and MVS contained in a dynamic or extended 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.
- Under CMS and MVS, the display of data can also be invoked automatically. If the run-time option ABSDUMP is in effect and your program abends (abnor-

mally terminates) in a program unit compiled with the SDUMP option or with the TEST option, all data in that program unit is automatically dumped. 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 of calling SDUMP from the main program and from a subprogram follow.

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 the subprogram 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. The saved value is 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.

If you are running under MVS, the value you supply to the SYSRCS subroutine is also returned as the condition code for the job step being processed if the program is terminated by a CALL EXIT or a CALL SYSRCX statement. The condition code is limited to decimal value 0 - 4095. If you code a value greater than 4095, the condition code returned may not match the value you specified in the SYSRCS subroutine.

Parallel Considerations: In parallel programs, the return code from this service subroutine has meaning only in the root task.

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.

Parallel Considerations: In parallel programs, the return code from this service subroutine has meaning only in the root task.

SYSRCX Subroutine

SYSRCX stops program processing using either the exit status previously set by SYSRCS, or the return code supplied in this call (*k*). The default value of the 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, control is returned to the operating system with the saved value. Any previously saved return code is ignored.

If you are running under MVS, the value you supply to the SYSRCX subroutine is also returned as the condition code for the job step being processed. The condition code is limited to decimal value 0 - 4095. If you code a value greater than 4095, the condition code returned may not match the value you specified in the SYSRCX subroutine.

If *k* is not specified and a return code was set by a previous call to SYSRCS, control is returned to the operating system with the saved return code. If the return code was not previously set, control is returned to the operating system with a return code of 0.

Parallel Considerations: In parallel programs, this service routine can be called only in serial parts of the root task. An execution time error will be detected if called from:

- a parallel loop
- a parallel section
- a subroutine invoked by PARALLEL CALL
- a subroutine executing in an originated task.

Termination Subroutines

These routines cause termination of the executing program.

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.

Parallel Considerations: In parallel programs, this service routine may be called only in serial parts of the root task. An execution time error will be detected if called from:

- a parallel loop
- a parallel section
- a subroutine invoked by PARALLEL CALL
- a subroutine executing in an originated task.

SYSABN/SYSABD Subroutines

The two abend routines allow you to specify abnormal termination of a 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. On MVS, 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.)

File Service Subroutines

These subroutines provide services to supply file characteristics, to identify valid unit numbers, and to manage file history for cross-language files.

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* [, *cpram1*, *value1*, *cpram2*, *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
64	Parameter BUFNO has an incorrect value
68	Parameter DEVCNT has an incorrect value

cpram

a character expression whose value when any trailing blanks are removed can be one of the following:

BLKSIZE	DEVICE	SECOND
BUFNO	DIR	VOLCNT
CYL TRK MAXBLK MAXREC	LRECL	VOLSER
DEVCNT	RECFM	VOLSERS

value

either a character expression, a character array, or an integer expression, depending on the parameter to which it corresponds.

DEVCNT The *value* that corresponds to DEVCNT is an integer expression of length 4. The expression's value specifies a maximum number of devices for a data set (valid number is from 1 to 59).

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 | VOLSERS

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 is used when there are more than one volume serial numbers needed. The maximum number of serial numbers is 255. 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.

- 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 for VS or VBS record format.
- 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.

BUFNO The *value* that corresponds to BUFNO is an integer expression of length 4. The expression's value is the number of I/O buffers to be used for a specified DASD unit. The valid buffer range is from 1 to 255. Values of less than one or greater than 255 are not allowed. If BUFNO is not specified, the default value is 2.

Programming Considerations for FILEINF

The parameters can be specified in any order and the CALL can contain none, some or all of the parameters. 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.

CMS Considerations: Only the following parameters are applicable to CMS: 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.

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 )
```

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 Fortran unit number that does not have a user-specified file definition associated with it (for example, with a ddname of FTnnF001 or FTnnK01, 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 <i>startnum</i> exceeds the largest allowable unit number, or is a negative value.
20	The value specified for <i>endnum</i> exceeds the largest allowable unit number.
24	The value specified for <i>endnum</i> is smaller than the value specified for <i>startnum</i> .

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 standard I/O unit number for the error messages and PRINT/WRITE statements (usually unit 6) will never be returned by UNTNOFD.

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 an available unit regardless of the file definitions in effect.

Syntax

CALL UNTANY (*r*code, *start*num, *end*num, *unit*num)

*r*code

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 <i>start</i> num exceeds the largest allowable unit number, or is a negative value.
20	The value specified for <i>end</i> num exceeds the largest allowable unit number.
24	The value specified for <i>end</i> num is smaller than the value specified for <i>start</i> num.

*start*num

a four-byte integer expression that specifies the first unit number of a range of unit numbers. The value of *start*num must be zero or a positive integer less than the largest unit number set up by the installation.

*end*num

a four-byte integer expression that specifies the last unit number of a range of unit numbers. The value of *end*num may be one of the following:

- a positive integer greater than *start*num 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.

*unit*num

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 *start*num and *end*num. If no unit number within the range meets the criteria, *unit*num becomes undefined.

UNTANY will never return the standard I/O unit number for the error messages and PRINT/WRITE statements (usually unit 6).

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.

Example:

The following

```
CALL UNTANY (IRCODE, 0, -1, IUNIT)
```

IGNFHU/IGNFHDD Subroutines

The file history subroutines allow you to identify the Fortran unit numbers and ddnames for which file history is to be ignored in determining file existence. These subroutines are used only with multi-language applications which manipulate a file within both the Fortran and non-Fortran portions. The file must not be currently open when the subroutine is called. See the VS FORTRAN Version 2 Programming Guide for usage and examples of these routines.

IGNFHU

IGNFHU causes the history of a file to be disregarded in determining file existence when the file has one of the following default ddnames:

FTxxF001	FTSQxxxx
FTxxP001	FTSRxxxx
FTxxK01	FTUPxxxx
FTPRTsss	FTUQxxxx
FTSPxxxx	FTURxxxx

where

xx is the unit number *unitnum* as below

sss

is the MTF subtask number

xxxx

is the task identifier

Syntax

CALL IGNFHU (*unitnum*, *rcode*)

unitnum
a four-byte integer expression that specifies the unit number. The value of *unitnum* must not be negative and must be less than or equal to the largest number set by the installation.

rcode
a four-byte integer variable or array element that will contain the return code upon return from IGNFHU.

Return Code	Explanation
0	Successful completion.
4	<i>unitnum</i> outside of allowable range.
8	The argument list has more than two arguments.
16	The file associated with the given unit is currently open.

If one or no arguments are specified, the call will be ignored.

Example:

The following call will cause the file history for unit 34 to be ignored.
CALL IGNFHU (34, Irc)

IGNFHDD

IGNFHDD causes the history of a file with a user-specified ddname to be disregarded in determining file existence.

Syntax

CALL IGNFHDD(*ddname*, *rcode*)

ddname
is a character expression whose value, when any trailing blanks are removed, is the ddname of the file definition that refers to the file. The value must be 1 to 8 characters.

rcode
a four-byte integer variable or array element that will contain the return code upon return from IGNFHDD.

Return Code	Explanation
0	Successful completion.
8	The argument list has more than two arguments.
12	<i>ddname</i> is invalid.
16	The file associated with <i>ddname</i> is currently open.

If one or no arguments are specified, the call will be ignored.

The following call will cause the file history for the file associated with `ddname` "MYFILE" to be ignored.

```
CALL IGNFHDD ("MYFILE", Irc)
```

Date/Time Subroutines

These subroutines provide various date and time services.

CLOCK/CLOCKX Subroutines

The `CLOCK` and `CLOCKX` subroutines provide values derived from the time-of-day clock that let you determine the elapsed time of a program or a portion of a program.

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` conforms to Industrial Real Time Fortran (IRTF) standards.

Syntax

```
CALL CLOCK (cpuclk [,count [,max ] ])
```

cpuclk

an integer variable or array element of length 4 in which the value derived from the time-of-day clock is returned in seconds. The time-of-day clock cycle starts on January 1, 1900.

count

an integer variable or array element of length 4 in which is placed the amount by which the time-of-day clock value is increased per second. *cpuclk* is incremented every second, so *count* is always 1.

max

an integer variable or array element of length 4 in which is placed the maximum value derived from the time-of-day clock. (In hexadecimal, this value is 7FFFFFFF.)

CLOCKX

Syntax

```
CALL CLOCKX (cpuckx [,xcount [,xmax ] ])
```

cpuckx

a real variable or array element of length 8 in which is returned an abbreviated version of the time-of-day clock, in microseconds. The cycle starts on January 1, 1900.

xcount

a real variable or array element of length 8 in which is placed the amount by which the time-of-day clock value is increased per second. *cpuclockx* is incremented every microsecond, so *xcount* is always 1000000.

xmax

a real variable or array element of length 8 in which is placed the maximum value derived from the time-of-day clock. (In hexadecimal, this value is 4D7FFFFFFFFFFFF0.)

Programming Considerations for CLOCK/CLOCKX:

Calls to CLOCK or CLOCKX subroutines are used to determine the elapsed time since a previous call to CLOCK or CLOCKX was made. To obtain the elapsed time used in a portion of a program, simply follow these steps:

1. Code two CLOCK or CLOCKX 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 elapsed time used by a portion of a program follows.

Using CLOCKX Calls:

```

Column: 1      6
-----
      REAL*8 START,TIME
      CALL CLOCKX(START)
      ...
      program code
      ...
      CALL CLOCKX(TIME)

      WRITE(*,100) (TIME-START)/1000000.
100    FORMAT( ' TIME(SECONDS)= ',D16.6)

```

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 CMS, the CPU time returned by the subroutine is in CMS virtual CPU time; that is, the time used directly by the user's virtual machine. On MVS, the CPU time returned by the subroutine is in MVS task time; that is, the time accumulated while an MVS task is in execution.

*r*code

an integer variable or array element of length 4 in which is placed the return code upon return from CPUTIME.

Parallel Considerations: This service subroutine is not allowed in a parallel program. A return code of 16 is returned when it is called in a parallel program.

Return Code	Explanation
0	Successful completion. The value in <i>accumcpu</i> can be used for either of the following: <ul style="list-style-type: none"> 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 <i>accumcpu</i> 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. <i>accumcpu</i> 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.
16	The subroutine was called in a parallel program.

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_B - ACCUM_A
          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 date and time that is available to the processor. To obtain date and time information, you need to call either DATIM or DATIMX. The time value is precise to the nearest hundredth of a second for MVS and to the nearest second for CMS.

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)

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 the moved characters are placed.

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 45 shows how ASSIGNM pads the output string and Figure 46 on page 342 shows how ASSIGNM truncates the output string.

1 6

```

-----
      CHARACTER*10 DBCS STUFF
      CHARACTER*15 STUFF
      INTEGER*4    RTCODE,RSNCODE
      DBCS STUFF = 'HOUSE k k'
      CALL ASSIGNM (DBCS STUFF,STUFF,RTCODE,RSNCODE)
*   These are the results of the CALL:
*   STUFF = HOUSE k k.....
*   (where "." is a blank character)
*   RTCODE = 0
*   RSNCODE = 0                (indicates padding with EBCDIC blanks)

```

Figure 45. Example of ASSIGNM Padding

```

1      6
-----
      CHARACTER*24 DBCS STUFF
      CHARACTER*14 STUFF1
      CHARACTER*11 STUFF2
      INTEGER*4    RTCODE,RSNCODE
      DBCS STUFF = 'WHAT A LIFE'
      CALL ASSIGNM (DBCS STUFF,STUFF,RTCODE,RSNCODE)
*   These are the results of the CALL:
*   STUFF1 = 'WHAT A'
*   RTCODE = 4
*   RSNCODE = 3                (indicates truncating)
*
      CALL ASSIGNM (DBCS STUFF,STUFF2,RTCODE,RSNCODE)
*   These are the results of the CALL:
*   STUFF2 = 'WHAT'.
*   (where "." is a blank character)
*   RTCODE = 4
*   RSNCODE = 3                (indicates truncating)

```

Figure 46. Example of ASSIGNM Truncation

ARGSTR Subroutine

The ARGSTR routine retrieves the user-supplied parameters from the command line.

Syntax

CALL ARGSTR(*string*,*rc*)

string

is a character variable, character substring, or character array element into which the user-specified run-time argument string will be placed.

rc is an INTEGER*4 variable or array element which will contain the return code from the ARGSTR service routine.

The return codes from ARGSTR are:

- 0** the string of user-specified run-time arguments is returned.
- 4** no user-specified run-time arguments are present; *string* is set to an all-blank string.
- 8** the size of *string* is not sufficient to contain all of the parameter string; only the portion of the string that can fit in *string* is returned. (Truncation occurs on the right.)

If *string* is longer than the run-time parameter string, blanks will be inserted on the right.

The run-time parameters are specified as the string of characters following the / in the invocation parameter string that starts the user program; all characters after the / through the end of the options comprise this string.

MVBITS Subroutine

The MVBITS routine allows a bit subfield of one integer value to be assigned to a bit subfield of another integer value.

Syntax

CALL MVBITS(*arg1*,*arg2*,*arg3*,*arg4*,*arg5*)

arg1

is an INTEGER*4 variable from which the source bit subfield is obtained.

arg2

is an INTEGER*4 variable or constant defining the beginning bit position in *arg1*. The bits are numbered 0 to 31 from right to left; the value must be in the range $0 \leq \text{arg2} \leq 31$.

arg3

is an INTEGER*4 variable or constant defining the number of bits to be processed; the value must be in the range $0 \leq \text{arg3} \leq 32$.

The sum of *arg2* and *arg3* must not be greater than 32.

If *arg3* is 0, no bits will be moved, but *arg2* and *arg5* will be checked for validity.

arg4

is an INTEGER*4 variable or constant into which the source bit subfield is placed. Bits not overlaid by the source bit subfield are not changed.

arg5

is an INTEGER*4 variable or constant defining the beginning bit position in *arg4*. The bits are numbered 0 to 31 from right to left; the value must be in the range $0 \leq \text{arg5} \leq 31$.

The sum of *arg2* and *arg5* must not be greater than 31.

Chapter 9. Routines for Use with Parallel Programs

This chapter describes routines you can use to control the execution of a parallel program. Parallel lock routines, parallel event routines, and a virtual processor query routine are provided.

Parallel event routines are provided to create parallel events that are used to synchronize the execution sequences of the primary parallel threads. A primary parallel thread is the thread that begins the program execution in the root task, or the initial thread is identified by the SCHEDULE statement. Parallel thread code can call event routines to signal completion of dependent code or signal a wait to wait until dependent code has been completed.

Parallel lock routines can be used to control access to shared data and resources. Shared resources include a parallel task's I/O units, error options table, and library service routines that operate on these resources.

A virtual processor query function is provided to allow the program to determine the number of virtual processors specified at run time.

The following parallel lock and event routines provide support for parallel processing of programs running under CMS and MVS.

Parallel Event Routines

PEORIG	Creates an event.
PEPOST	Posts an event.
PEWAIT	Causes the routine to wait until an event has completed.
PETERM	Deletes an event.

Parallel Lock Routines and Function

PLORIG	Creates a lock.
PLLOCK	Obtains a lock.
PLCOND	Obtains a lock if it is available.
PLFREE	Releases a lock.
PLTERM	Deletes a lock.

Virtual Processor Query

NPROCS	Determines the number of virtual processors specified at run time.
--------	--

Parallel Execution Environment Control Routines

PFAFFS	Starts affinity mode
PFAFFC	Releases affinity mode
PYIELD	Yields control of parallel environment

Parallel Trace Routines

PTWRIT	Generates user-defined trace records
--------	--------------------------------------

Parallel Event Services

A parallel event is a synchronization facility that provides a variety of ways to control synchronization of primary parallel threads. These threads include the root task, primary thread, and those threads identified by the SCHEDULE statements. Parallel events cannot be used within PARALLEL DO or PARALLEL SECTIONS constructs, or the subroutines invoked by PARALLEL CALL statements. A parallel event must be originated and terminated in the same parallel task.

You can use parallel event services to synchronize dependent code between parallel threads. One or more parallel threads can signal the occurrence of an event to other parallel threads by posting the event. Other parallel threads can synchronize with the event by waiting for the event cycle to complete.

Synchronization is accomplished by halting execution of parallel threads that are waiting for an event cycle to complete. An event cycle is completed when a specified number of post signals and a specified number of wait signals have been received for a particular event. The waiting threads are restarted after event cycle completion.

The PEORIG event routine is used to define parallel events. Each definition includes specification of an eventid that will allow selection of a particular event when more than one event is active. Event cycle parameters, postcount and waitcount, are also specified as part of the event definition. The event definition can also require post signals to come from unique parallel threads.

The PEPOST event routine is used to send post signals to a particular event. You can use the PEPOST routine to signal the completion of dependent code by one parallel thread to one or more other parallel threads. If the event cycle requires unique posts and a parallel thread signals a second post before the event cycle is complete, it is suspended until the event cycle is complete.

The PEWAIT event routine is used by a parallel thread to halt execution of the thread until a particular event cycle is complete.

Parallel Event Cycle Operations

Event cycle parameters (postcount and waitcount) are specified as arguments of the PEORIG call. If only one of these counts is specified with a positive value, the event cycle will be completed when the number of that type of signal (post or wait) reaches the value defined (postcount or waitcount). When both postcount and waitcount are defined as positive numbers, the event cycle is completed when post signals from parallel threads reach the postcount value and the number of waiting threads reaches the waitcount value. The user has the option of requiring each poster to be unique, or allowing the same parallel thread to post the event multiple times.

Figure 47 on page 346 summarizes event cycle operations based on how the event cycle was defined.

Figure 47. Parallel Event Cycle Operations

postcount	waitcount	unique or non-unique posts	event cycle determined by
-1	>0	either	waitcount
0	>0	not used	waitcount
>0	-1	either	postcount
>0	0	unique	postcount
>0	>0	either	postcount and waitcount

PEORIG Routine

The PEORIG routine creates a parallel event, initializes an event cycle, and returns an event identifier.

Syntax

```
CALL PEORIG (eventid [,postcount [,waitcount [,unique ]]] )
```

eventid

is an integer variable or an array element of length 4 in which the event identifier is returned.

The event identifier is for identification purposes only; its value should not be altered or used for computation.

The *eventid* variable name is used in subsequent PEPPOST and PEWAIT routine calls by parallel threads when a particular event is needed to synchronize execution. If the PEPPOST and PEWAIT calls are made from scheduled routines, then the *eventid* variable must be made available for subsequent calls to PEPPOST or PEWAIT.

postcount

is an integer expression of length 4 that specifies the number of post signals that are required to complete a cycle of the event. The *postcount* value, if specified, must be one of the following:

- 1** Any number of post signals are permissible in a cycle. A positive waitcount must be specified to define the event cycle.
- 0** Post signals are not required. PEPPOST calls for the particular event will be ignored. A positive waitcount must be specified to define the event cycle.
- >0** Specifies the number of post signals required to end an event cycle. If waitcount is also positive, the end of an event cycle occurs when the number of wait and post signals reach their specified count.

The default value for *postcount* is 1.

waitcount

is an integer expression of length 4 that specifies the number of wait signals that are required to complete a cycle of the event. The *waitcount* value, if specified, must be one of the following:

- 1** Any number of wait signals are permissible. Each wait signal will cause the calling thread to be suspended until the end of the event cycle. A positive postcount must be specified to define the event cycle.
- 0** No wait signals are required. PEWAIT calls for this event will be ignored. A positive postcount must be specified to define the event cycle.
- >0** Specifies the requisite number of wait signals to complete an event cycle. If postcount is also positive, the end of an event cycle occurs when both post and wait signals reach their specified counts. Each wait signal suspends the calling thread until the end of the event cycle.

The default value for *waitcount* is 1.

unique

is an integer expression of length 4 that specifies whether or not post signals need to come from different parallel threads during an event cycle.

- 0** All post signals sent to an event by any parallel thread are counted to reach the end of an event cycle.
- 1** Each parallel thread can send a post signal once during an event cycle. A duplicate signal received from a thread prior to post and wait signals ending a cycle causes the posting thread to be suspended until the end of the cycle. The duplicate signal will be applied to the next event cycle.

The default value for *unique* is 1.

Note: At least one of the values for *postcount* or *waitcount* must be positive. An error is detected if this condition is not met.

Example 1:

```
CALL PEORIG(IEVENT3)
```

Example 2:

```
CALL PEORIG(IEVENT4,1,1,1)
```

Example 3:

```
CALL PEORIG(IEVENT5,2)
```

Example 4:

```
CALL PEORIG(IEVA,2,3)
```

PEPOST Routine

The PEPOST routine posts the specified event. Once the postcount has been reached, any additional post signals sent to the event are applied to the next cycle. In addition, the threads that send the additional post signals are suspended until the next event cycle begins. For information on postcount, refer to “PEORIG Routine” on page 346.

Syntax

```
CALL PEPOST (eventid)
```

eventid

is an integer variable or an array element of length 4 that contains the identifier of the event to be posted.

An error is detected when the calling program attempts to post an event that has not been originated.

Example:

```
CALL PEPOST(EVENT3)
```

PEWAIT Routine

The PEWAIT routine suspends the calling parallel thread until the event cycle is complete. Once the waitcount has been reached, any additional wait signals sent to the event are applied to the next cycle. In addition, the threads that sent the additional wait signals are suspended until the next event cycle ends. For information on *waitcount*, refer to “PEORIG Routine” on page 346.

Syntax

```
CALL PEWAIT (eventid)
```

eventid

is an integer variable or an array element of length 4 that contains the identifier of the event to be waited on.

An error is detected when the calling program attempts to wait on an event that has not been originated.

Example:

```
CALL PEWAIT(EVENT3)
```

PETERM Routine

The PETERM routine deletes the specified event.

Syntax

```
CALL PETERM (eventid)
```

eventid

is an integer variable or an array element of length 4 that contains the identifier of the event to be deleted.

An error is detected when the calling program attempts to:

- Delete an event that has not been originated.
- Delete an event from a parallel task that did not originate the event.
- Delete an event in the middle of a cycle.

Example:

CALL PETERM(EVENT3)

Parallel Event Example

Figure 48 is an example of how an event is defined and used to synchronize three parallel threads for an event cycle of two post signals and one wait signal. This example shows how parallel threads are suspended and resumed in this event cycle.

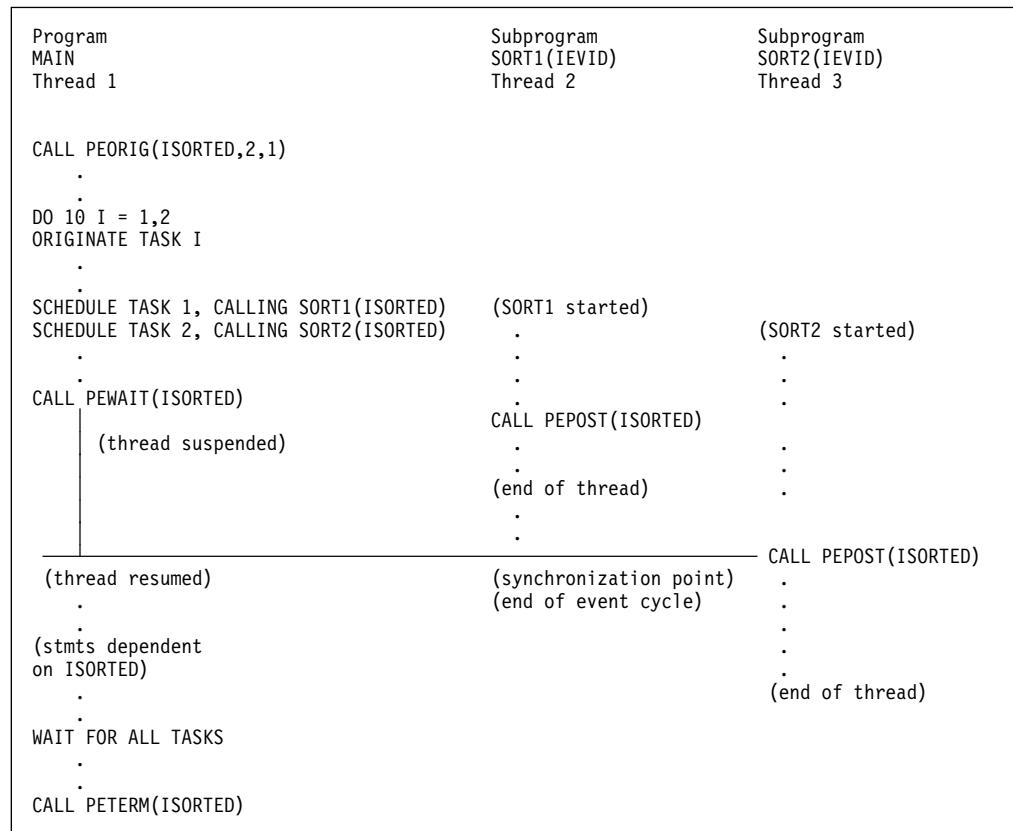


Figure 48. Parallel Event Example

Parallel Lock Services

You can use parallel lock services to control access to shared data and to synchronize changes when updating shared data areas. Parallel locks can also be used to control critical sections of code, where a user wants to ensure that only one parallel thread at a time executes the section. Lock routines are provided to create, obtain, release, and terminate parallel locks. Two routines for obtaining locks are provided.

One routine returns only when the lock is obtained, which might require waiting for one or more parallel threads to complete data updates and release the lock. Another lock routine can be called to obtain the lock if it is available, or set a non-zero function value if it is not available. Locks can be obtained in shared or exclusive use modes.

One thread can update a shared variable for multiple threads to reference. The update must be done only by one thread at a time, and thus exclusive mode is provided. Once the update is complete, multiple threads can reference it at a time. To reference the data, they must still use a lock to ensure that the variable is not being updated at the time. Shared mode is provided to allow multiple concurrent references.

The PLORIG lock routine is used to create a lock and return an identifier for the lock. After creating the lock, any parallel thread with access to the lock identifier can obtain shared or exclusive use of the lock for referencing or updating shared data variables.

The PLLOCK lock routine is used to obtain a parallel lock in either shared or exclusive mode. If the lock is being used by another parallel thread in an exclusive mode, the requesting parallel thread will be suspended until the lock has been released by the exclusive user. The lock is then obtained for the waiting parallel thread in either a shared or exclusive mode, and the thread is resumed. If the lock is requested in the shared mode and other parallel threads are using the lock in a shared mode, then the lock will be obtained without waiting. The call to PLLOCK must identify any variables shared with other parallel threads that are referenced within the locked code to ensure their current shared values are used. The user should reference or update the lock controlled variables in as few operations as possible and then release the lock to avoid excessive time waiting for locks.

The PLCOND function can also be used to request a lock. If the lock is available, the lock is obtained for the requesting parallel thread, and the function value is set to 0. If the lock has not been obtained, the PLCOND routine returns immediately back to the calling thread with a non-zero function value. You must test the function value of PLCOND to determine if the lock was obtained so that your lock dependent code can be run or some other action must be taken. If the lock was obtained by the PLCOND function, code that references or updates shared variables may be run. The code must then release the lock with PLFREE. If the lock was not available, you can execute other code that is not dependent on obtaining the lock before making another lock request. This technique provides a method for dynamically balancing concurrent execution of parallel threads.

The PLFREE lock routine is used to release the lock for use by other parallel threads. The PLFREE call must identify updated variables to be shared with other threads.

The PLTERM routine is used to terminate a lock. A parallel lock must be originated and terminated in the same parallel task. Originate and terminate calls can be made from the same or different parallel threads in the parallel task.

Parallel Lock Contention

Lock request contention will be minimized if shared lock modes can be used. Whenever locks are used, it is best to make sure that only a minimum set of operations is performed while the lock is held.

Figure 49 shows the possible lock contention based on the lock request and the lock mode status. The “wait for lock” means the parallel thread making the request will be suspended until the lock is released by the parallel thread holding the lock. The function value (fv) represents the value returned from the function subprogram PLCOND.

lock request	Lock Mode Status		
	free	shared	exclusive
PLLOCK shared mode	obtained	obtained	wait for lock
PLLOCK exclusive	obtained	wait for lock	wait for lock
PLCOND shared mode	obtained fv=0.0	obtained fv=0.0	not obtained fv=-0.0
PLCOND exclusive	obtained fv=0.0	not obtained fv=-0.0	not obtained fv=-0.0

Figure 49. Parallel Lock Contention Possibilities

PLORIG Routine

The PLORIG routine creates and initializes a lock and returns an identifier for the lock.

The identifier is used in subsequent calls to lock request routines PLLOCK and PLCOND, and the lock release routine PLFREE. If the subsequent calls to obtain or release the lock are made from routines other than the routine originating the lock, the lock identifier must be passed as a routine argument or shared as a common variable.

Syntax

```
CALL PLORIG (lockid)
```

lockid

is an integer variable or an array element of length 4 in which the lock identifier is to be returned. The lock identifier is for identification purposes only. Its value should not be used for computation.

Example:

```
CALL PLORIG(LOCK3)
```

PLLOCK Routine

The PLLOCK routine is used to obtain a lock in exclusive or shared mode. If another parallel thread is using the lock, the parallel thread calling PLLOCK may have to wait for the lock. Figure 49 on page 351 defines which conditions will require waiting. For example, if the parallel threads are only using the lock in shared mode and the calling thread requests shared mode, the lock will be obtained without any wait.

Note: The lock identifier and the shared variables must be variables that are available in the thread calling the PLLOCK routine.

Syntax

```
CALL PLLOCK (lockid [,mode [,var [,var]... ] ])
```

lockid

is an integer variable or an array element of length 4 that contains the identifier of the lock to be obtained.

mode

is an integer expression of length 4 that specifies the mode of the lock to be obtained.

0 The lock is requested in exclusive mode. (This is the default).

-1 The lock is requested in shared mode.

var

is the name of a variable, array, or array element whose access must be synchronized with other parallel threads. When items on this list are referenced after the CALL PLLOCK, they are fetched from storage. Thus, changes made in other parallel threads are communicated to the current parallel thread. Items in COMMON are automatically fetched after a CALL, so that they need not be identified on the PLLOCK statement.

It is an error to:

- Request a lock that has not been originated.
- Request a lock that had already been obtained by the parallel thread but not released.

Example 1:

```
SUBROUTINE SORT2Z(ILOCK4)
COMMON / ASHR/IVAR1

CALL PLLOCK(ILOCK4,-1)      ! Obtain shared lock, shared variable
                             ! IVAR1 a COMMON variable
      IF (IVAR1.EQ.5) SVAR=0.0 ! SVAR is not shared
CALL PLFREE(ILOCK4)         ! Release lock, no changed variables
```

Example 2:

```

SUBROUTINE SORT1(ILOCK4,IVAR2)
CALL PLLOCK(ILOCK4,0,IVAR2) ! Obtain exclusive lock, shared variable
                                ! IVAR2 is accessed in the lock code and
                                ! is not a COMMON variable

    IVAR2=IVAR2+1
    SVAR=2.0*IVAR2           ! SVAR is not shared
CALL PLFREE(ILOCK4,IVAR2)   ! Ensure non-COMMON variable is stored
                                ! before lock is released

```

PLCOND Function

The PLCOND function conditionally obtains the specified lock. Unlike PLLOCK, PLCOND does not wait to obtain the lock before it returns. It returns an indication of whether or not the lock was obtained.

Syntax

PLCOND (*lockid* [,*mode* [,*var* [,*var*]...]])

Implicitly, PLCOND returns a real value of length 4. PLCOND may be specified as integer, in which case the value returned is an integer of length 4.

If the return value is 0, then the lock was obtained. If the return value is not 0, then the lock was not obtained.

lockid

is an integer variable or an array element of length 4 that contains the identifier of the lock to be obtained.

mode

is an integer expression of length 4 that specifies the mode of the lock to be obtained.

0 The lock is requested in exclusive mode. (This is the default.)

-1 The lock is requested in shared mode.

var

is the name of a variable, array, or array element whose access must be synchronized with other parallel threads. After the PLCOND function is invoked, items on the list are fetched from storage when they are referenced. Thus, changes made to shared data areas by other parallel threads may be communicated to the current parallel thread.

Items in COMMON are automatically fetched after a CALL, so that they need not be identified on the PLCOND statement.

It is an error to:

- Request a lock that has not been originated.
- Request a lock that has already been obtained by the parallel thread and not released.

Example:

```

IF (PLCOND(ILOCK,0,IVAR,RVAR) .EQ. 0.0) THEN
  IF (IVAR.EQ.1) JVAR=0
  RVAR=RVAR+2.0
  CALL PLFREE(ILOCK,RVAR)
  CALL SUBB                ! code not dependent on IVAR
ELSE
  CALL SUBB                ! code not dependent on IVAR
  CALL PLLOCK(ILOCK,0,IVAR,RVAR)
  IF (IVAR.EQ.1) JVAR=0
  RVAR=RVAR+2.0
  CALL PLFREE(ILOCK,RVAR)
ENDIF

```

PLFREE Routine

The PLFREE routine is used to release the specified lock for use by one or more parallel threads.

Syntax

CALL PLFREE (*lockid* [,*var* [,*var* [,*var*...]]])

lockid

is an integer variable or array element of length 4 that contains the identifier of the lock to be released.

var

is the name of a variable, array, or array element not in shared common whose changes must be synchronized with other parallel threads. Items on this list are stored before the lock is released. Note that items in COMMON are automatically stored prior to a CALL, so that they need not be identified on the PLFREE statement.

It is an error to:

- Release a lock that has not been originated.
- Release a lock not previously obtained by the current parallel thread.

Note: If a parallel thread obtains a lock and does not release it before the thread finishes processing, the lock will be automatically released.

Example 1:

```
CALL PLFREE(LOCK3)
```

Example 2:

```
CALL PLFREE(LOCK3,VAR1,VAR2)
```

PLTERM Routine

The PLTERM routine deletes the specified lock.

Syntax

```
CALL PLTERM (lockid)
```

lockid

is an integer variable or an array element of length 4 that contains the identifier of the lock to be deleted.

It is an error to:

- Delete a lock that has not been originated.
- Delete a lock from a parallel thread that is not in the same parallel task as the parallel thread that originated the parallel lock.
- Delete a lock that has been obtained and not yet released.

Example:

```
CALL PLTERM(LOCK3)
```

Parallel Lock Example

Figure 50 on page 356 is an example of how a lock is defined and used to synchronize access to shared data variables by three parallel threads.

Notice the following:

1. Thread 1 has LID in exclusive mode.
2. Thread 2 is suspended when it asks for shared access to LID, because Thread 1 already has LID in exclusive mode.
3. Thread 1 releases LID with the call to PLFREE.
4. Thread 2 resumes execution holding the lock LID in shared mode.
5. Thread 3 asks for the lock and is able to obtain the lock at the same time as Thread 2, since they both have asked for it in shared mode.

Program MAIN Thread 1	Subprogram SORT3(LID,VAR1) Thread 2	Subprogram SORT4(LID,VAR1) Thread 3
CALL PLORIG(LID)		
.		
PARALLEL CALL SORT3(LID,VAR1)	(SORT3 started)	
PARALLEL CALL SORT4(LID,VAR1)	.	(SORT4 started)
.	.	.
CALL PLLOCK(LID,0,VAR1)	.	.
(lock obtained for update)	.	.
VAR1=VAR1+5	CALL PLLOCK(LID,-1,VAR1)	.
.	(thread suspended, lock	.
CALL PLFREE(LID,VAR1)	in exclusive use)	.
(release lock, identify		.
VAR1 as update variable)	(lock obtained for sharing,	.
.	thread resumed)	.
.	SVAR=VAR1	CALL PLLOCK(LID,-1,VAR1)
.	.	(lock obtained for sharing)
.	CALL PLFREE(LID)	XVAR=VAR1
.	.	.
.	(end of thread)	CALL PLFREE(LID)
WAIT FOR ALL CALLS		.
.		(end of thread)
CALL PLTERM(LID)		

Figure 50. Parallel Lock Example

Virtual Processor Query

NPROCS Function

NPROCS is a function that can be used to determine the number of virtual processors that were specified at run time.

The NPROCS function returns the number of virtual processors specified on the PARALLEL run-time option when the program started, or returns the value of **1** if the parallel option was not specified.

Syntax

NPROCS ([*n*])

n is an integer variable or an array element of length 4. The value returned to *n* is the number of virtual processors, and that value is an integer. This value is also the function value.

Example 1:

```
I = NPROCS()
```

Example 2:

```
I = NPROCS(J)           ! Both I and J are set to number of processors
```

Example 3:

```

      DO 10, K=1,NPROCS()
        ORIGINATE TASK K
10    CONTINUE

```

Parallel Execution Control

You can use library service routines to control the parallel execution environment. These user-affinity controls will in turn enable you to use assembler subroutines that invoke system services that require processor/task affinity.

VS FORTRAN Version 2 provides a "set affinity" service to change execution of the parallel environment to the main task/thread, and a "release affinity" service to re-enable full multi-tasking/thread execution.

VS FORTRAN Version 2 also provides a "yield" library service routine to allow you to temporarily yield execution for a given thread to enable other threads to execute. This function, similar to the MVS service CALLDISP, extends the multiprocessing capabilities to enable you to interrupt the execution of one thread and avoid interlocks, delays, and conflicts in the parallel environment.

PFAFFS Routine

Syntax

```
CALL PFAFFS
```

The PFAFFS routine switches the virtual processor the parallel thread is running on to the same one that the parallel program first started running on. Once running on this virtual processor, MVS or CMS system services can be invoked. The parallel thread will continue to run on the same virtual processor until the PFAFFC routine is called or termination occurs.

- Under MVS, the thread is executing under control of the TCB under which the VS FORTRAN library was initialized.
- Under CMS, the thread is executing under control of virtual processor 0 and has access to all CMS system functions, callable system services, and the CMS NUCON area.

This affinity remains in effect until terminated by calling the PFAFFC service routine. While the current thread is executing, any other threads that have not called PFAFFS can continue to execute concurrently.

PFAFFC Routine

Syntax

```
CALL PFAFFC
```

The PFAFFC routine releases any processor affinity previously set by calling PFAFFS.

Execution of the current thread then resumes in the normal parallel environment. The restrictions on execution in the parallel environment, with respect to using system services, are re-applied to the current thread.

PYIELD Routine

Syntax

CALL PYIELD

The PYIELD routine causes the VS FORTRAN Version 2 library to interrupt the execution of the current thread and attempt to execute any threads now waiting to execute.

By using the PYIELD service routine, the current thread voluntarily gives up control to (yields) the processor to another thread. Execution of the current (yielding) thread resumes when a processor becomes available to execute the thread; this occurs when another thread terminates or gives up control by calling PYIELD.

Parallel Trace Facility

You can record events occurring during parallel execution by the use of the trace facility for the parallel execution environment.

The Parallel Trace Facility generates trace records to identify the entry and exit from every subprogram, entry to the main program, termination of the main program, and each defined event occurring during the execution of a parallel program. (A *defined event* is the start of execution of a parallel section, or one or more iterations of a parallel loop known as a *chunk*.)

By using the data obtained via the Parallel Trace Facility, you can analyze the behavior of parallel programs and perform necessary tuning and debugging functions.

The Parallel Trace Facility records each significant action as a record in an output file, known as the **Trace File**. Separate programs, collectively referred to as *trace tools*, are used to analyze these records; trace tool programs must be used as post-processors against the Trace File after execution of the VS FORTRAN Version 2 parallel program has completed.

The VS FORTRAN Version 2 Parallel Trace Facility is similar to the Trace Facility provided in the VS Fortran Parallel PRPQ (Program Number 5799-CTX). If you are familiar with that facility, you will find that the Parallel Trace Facility is an extension of that facility; some elements have been changed and improved.

In order to use the Parallel Trace Facility, you must specify the PTRACE run-time option when executing VS FORTRAN Version 2 programs; see the VS FORTRAN Version 2 Programming Guide for additional information. VS FORTRAN Version 2 parallel programs compiled with the PARALLEL(TRACE) compiler option will generate additional information beyond that for normal programs via the Parallel Trace Facility. Program execution time increases slightly. The use and function of the PARALLEL(TRACE) compiler option is also explained in the VS FORTRAN Version 2 Programming Guide.

Program Tracing Categories: The significant actions for which tracing occurs in VS FORTRAN Version 2 parallel programs can be found in the *VS FORTRAN Version 2 Programming Guide*.

PTPARM Routine

Syntax

```
CALL PTPARM (argstring)
```

argstring

is a character constant, variable, substring, or array element with a blank-delimited value specification of which parallel trace options are to be applied; it is one or more of the values allowed as suboptions for the PTRACE run-time option (see the VS FORTRAN Version 2 Programming Guide).

The PTPARM service routine provides a means for your program to dynamically control the tracing activity of the Parallel Trace Facility during program execution.

By using the PTPARM service routine, you can turn tracing on and off for the executing program, or you can selectively turn on or off certain trace categories.

Example:

```
CALL PTPARM('NOCOMMON EVENTS TIMES')
```

PTWRIT Routine

The Parallel Trace Facility allows you to generate your own trace records for events that you determine are of significance in the execution of your program, through the PTWRIT service routine.

Syntax

```
CALL PTWRIT(category,type,user_data [,user_data_len])
```

category

is a character constant, variable, substring, or array element whose value is the name of the category for which the trace record is generated. It is one or more of the values allowed as suboptions for the PTRACE run-time option (see the VS FORTRAN Version 2 Programming Guide).

type

when *category* is 'USER', *type* is an INTEGER*4 variable or integer constant in the range of $0 \leq type \leq 255$, which is the user-determined record subtype. When *category* is other than 'USER', *type* is a character constant, variable, substring, or array element with the value 'USER'.

user_data

is a character constant, variable, substring, or array element whose value is the user-determined data to be placed into the trace record.

user_data_length

is an integer variable or constant that is the length of the data in the *user_data* field.

Normally *category* is the value 'USER', indicating that a trace record of category USER is to be created. However, this argument may be any of the valid trace record categories that are allowed for the PTRACE run-time option and the PTPARM service routine.

If the category specified is not allowed for tracing, the call to the PTWRIT service routine is ignored and no record is generated.

Regardless of the category identified in the call to the PTWRIT service routine, the trace record generated is always of type USER.

Argument *type* identifies the record subtype for the created trace record, when the category is USER. When the category is other than USER, *type* is assumed to be 'USER'.

For example, the call:

```
CALL PTWRIT ('USER',99,MYDATA,32)
```

generates a USER category trace record of subtype 99, whose data section contains the first 32 bytes of the character variable MYDATA.

However, if a trace record is to be created in the LOCKS category, then the call:

```
CALL PTWRIT ('LOCKS','USER',MYDATA,32)
```

causes this record to be written.

Chapter 10. Data-in-Virtual Subroutines

Data-in-Virtual Overview

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 and MVS/ESA. 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*. Also, the technical bulletin *An Introduction to Data-in-Virtual*, provides background information on using the data-in-virtual facility functions under MVS/XA.

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 or extended common to all or part of the data set. You can think of the common block 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, data-in-virtual brings only the parts of the data object that your program is currently mapping and actually refers to into virtual storage. When you want to save changes, data-in-virtual saves to permanent storage only those parts that your program actually changed.

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.

The SHRCOM subroutine can be used to designate a dynamic common block as shareable among the main task program and the parallel subroutines, and thus access to the data object may be supplied for all 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.

Using Data-in-Virtual in a Parallel Program

If you are running your applications in a parallel program, you must make all data-in-virtual subroutine calls from the primary thread in the root task.

In a parallel program, dynamic or extended commons can be designated as being shared across originated tasks via the SCHEDULE statement parameters, or shared with other parallel threads via the PARALLEL CALL or call or function reference in a parallel loop or section. See the SCHEDULE statement in the “Parallel Task Management Statements” on page 264.

When sharing a dynamic or extended common, do not have the root task reset, terminate, or remap the common until all subroutines have completed. Otherwise, the results will be unpredictable.

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 or extended common, the window, to a given data object (see Figure 51).

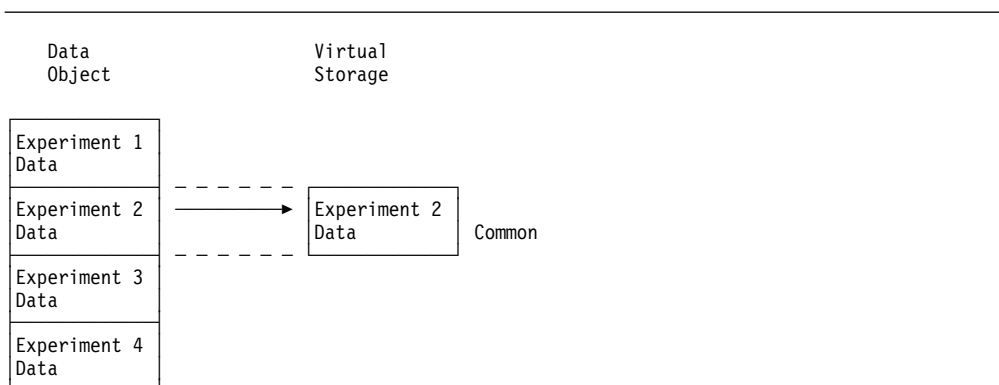


Figure 51. Fixed-View Method

With the varying-view method, you can map any number of dynamic or extended commons to the data object and can simultaneously view different parts of it (see Figure 52 on page 363).

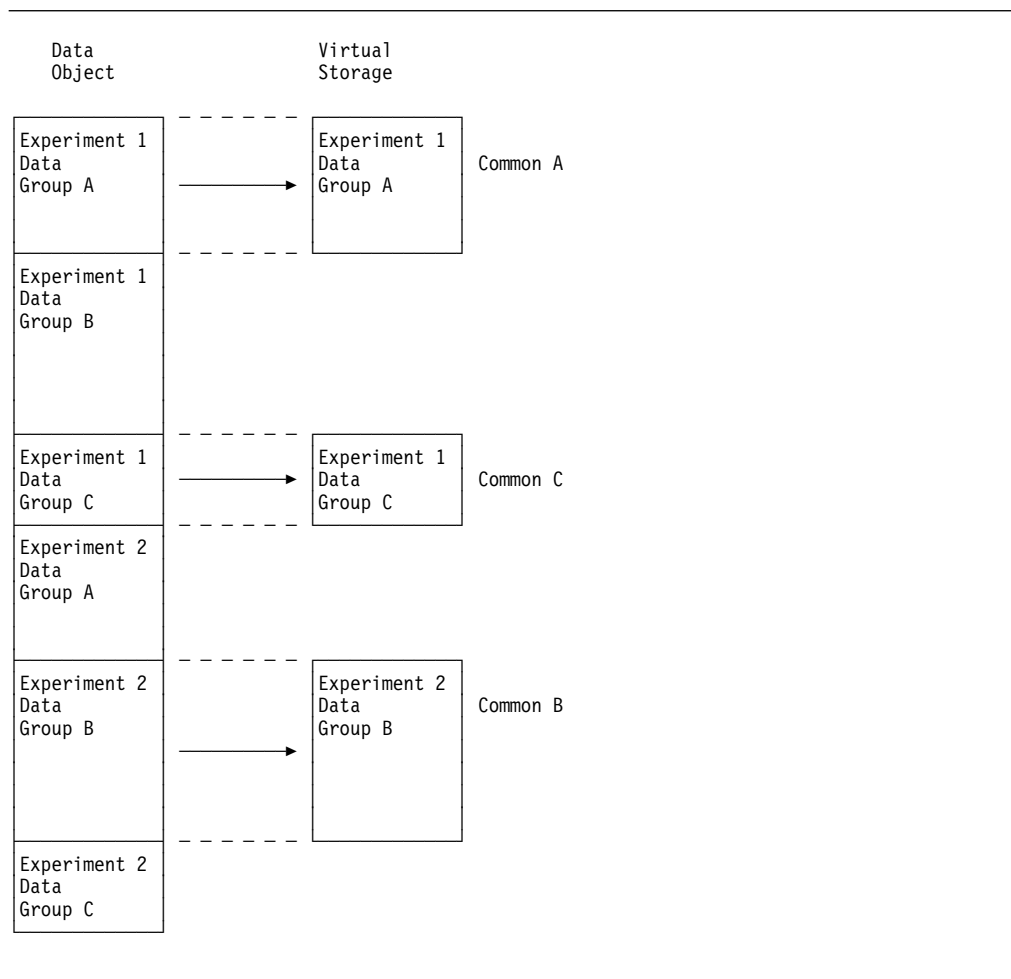


Figure 52. Varying-View Method

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 or extended 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 or extended 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 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 common is discussed under “Sample Program with Fixed-View Subroutines” on page 374, “Sample Program with Varying-View Subroutines” on page 376, and “Ensuring Data Integrity” on page 381.

Syntax of the Subroutines

The following sections give the syntax of the data-in-virtual subroutines:

- DIVINF, DIVVWF, and DIVTRF, for the fixed-view method
- DIVINV, DIVVWV, and DIVTRV, and DIVCML for the varying-view method
- DIVSAV and DIVRES, for both the fixed- and varying-view methods

Syntax of Fixed-View Subroutines

The fixed-view subroutines give you the ability to treat a data object as one or more instances of a single dynamic or extended common.

DIVINF Subroutine

The DIVINF subroutine allows you to associate a data object with a dynamic or extended common for reading or for reading and writing.

Syntax

CALL DIVINF (*r*code, *dyn*com, *obj*size_*commons*, *divobj*, *type*, *access*)

*r*code
is a four-byte integer variable or array element that will contain the return code from subroutine DIVINF.

Return Code	Explanation
0	Successful completion
4	Successful completion, but the length of the common is not an exact multiple of 4096. Unless you expect to modify the common and reuse the data object, you may ignore this return code.
8	Not a dynamic or extended common for <i>dyncom</i>
12	Not DDNAME, DSNAME or DSN for <i>type</i>
16	Not READ or READWRITE for <i>access</i> (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 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 or extended common, when trailing blanks are removed and when folded to uppercase.

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 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 uppercase, 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 lowercase will be folded to uppercase. 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 uppercase, 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 uppercase, 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:

A given data object may be associated with only one dynamic or extended common at a time, and a given dynamic or extended common may be associated with only one data object at a time. The DIVTRF subroutine must be called to disassociate a 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 or extended common will map.

Syntax

CALL DIVVWF (*r*code, *dyn*com, *map*num)

*r*code
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 or extended common for <i>dyn</i> com
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 <i>map</i> num should have been changed.
64	The specified common is not associated with a data object.
72	Zero or a negative value specified for <i>map</i> num
128	Data-in-virtual services failed

*dyn*com
is a character expression whose value is the name of a dynamic or extended common, when trailing blanks are removed and when folded to uppercase.

*map*num
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 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 51 on

page 362, 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 8192 and the value 3 indicates an offset at byte 16384.

Programming Notes:

A given dynamic or extended common can be mapped to only one data object at a time, and a given data object can be mapped to only one dynamic or extended common at a time.

If you request that a mapped 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 common via the DIVVWF call.

DIVTRF Subroutine

The DIVTRF subroutine terminates the association of the data object to the dynamic or extended 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 <i>dyncom</i>
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 or extended 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 or extended common, when trailing blanks are removed and when folded to uppercase.

Programming Note: DIVTRF must be called to terminate access to a data object before the same 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 or extended 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 (*r*code, *obj*-id, *obj*size_*pages*, *divobj*, *type*, *access*)

*r*code
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 <i>type</i>
16	Not READ or READWRITE for <i>access</i> (or blank with DDNAME <i>type</i>)
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 <i>divobj</i>
36	<i>Divobj</i> specified conflicts with <i>type</i> 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.

*obj*size_*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 uppercase, 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 lowercase will be folded to uppercase. 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 uppercase, is DDNAME, DSNNAME, 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. DSNNAME 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 uppercase, 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 or extended common that has access to another data object, an error results. The DIVTRV subroutine must be called before DIVINV to disassociate the common from the first data object.

Once the data object has been accessed through DIVINV, the DIVVWV subroutine must be 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 or extended common will map.

Syntax

```
CALL DIVVWV (rcode, dyncom, offset, obj-id)
```

*r*code

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 common is not an exact multiple of 4096. Unless you expect to modify the common and reuse the data object, you may ignore this return code.
8	Not a dynamic or extended common for <i>dyncom</i>
44	An attempt was made to use a 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 <i>obj-id</i> specified is not associated with any data object.
60	Negative value specified for <i>offset</i>
128	Data-in-virtual services failed

dyncom

is a character expression whose value, when any trailing blanks are removed and when folded to uppercase, is the name of the dynamic or extended common.

offset

is an integer expression containing the offset value in units of pages, at which the 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 portion 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 or extended 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 common.

Multiple calls to DIVVWV can be made on the same data object to simultaneously associate multiple commons with different parts of the data object. However, these commons may not overlap.

DIVCML Subroutine

The DIVCML subroutine obtains the length of a dynamic or extended 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 (*r*code, *dyn*com, *length*)

*r*code
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 common is not an exact multiple of 4096. Unless you expect to modify the common and reuse the data object, you may ignore this return code.
8	Not a dynamic or extended common for <i>dyn</i> com
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

*dyn*com
is a character expression whose value, when any trailing blanks are removed and when folded to uppercase, is the name of the dynamic or extended common.

length
is a four-byte integer variable or array element that will contain the returned length of the common in pages (4096 bytes per page). For example, if the common is 5000 bytes long, the returned *length* 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 55 on page 377 and Figure 56 on page 379.

DIVTRV Subroutine

The DIVTRV subroutine terminates the association between the data object ID and the data object.

Syntax

CALL DIVTRV (*r*code, *obj*-id)

*r*code
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 <i>obj</i> -id
128	Data-in-virtual services failed

obj-id
is a character variable or array element of at least eight bytes in length 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 or extended commons that are mapped to a data object when a DIVTRV call is made will automatically unmap.

Syntax of Common Subroutines

The DIVSAV and DIVRES subroutines can be used with both the fixed-view and varying-view methods.

DIVSAV Subroutine

The DIVSAV subroutine saves changes made in the dynamic or extended common to the data object that has been accessed for READWRITE.

Syntax

CALL DIVSAV (*r*code, *dyn*com)

*r*code
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 <i>dyncom</i>
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 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 or extended common, when trailing blanks are removed and when folded to uppercase.

DIVRES Subroutine

The DIVRES subroutine resets the data in the dynamic or extended common to the values in the mapped part of the data object, eliminating any changes that have been made in the common, either initially or since the last DIVSAV.

Syntax

CALL DIVRES (*r*code, *dyncom*)

*r*code

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 <i>dyncom</i>
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 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 or extended common, when trailing blanks are removed and when folded to uppercase.

Interface to the Data-in-Virtual Functions

Figure 53 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*.

Figure 53. Data-in-Virtual Subroutines			
FIXED-VIEW SUBROUTINES	VARYING-VIEW SUBROUTINES	DESCRIPTION	DIV FUNCTION
DIVINF	DIVINV	Associates the data object with a dynamic or extended 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 or extended 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 or extended common to the data object.	SAVE
DIVRES	DIVRES	Resets the dynamic or extended 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 or extended 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 or extended common.	None

Sample Program with Fixed-View Subroutines

The use of the fixed-view subroutines is shown in Figure 54 on page 375. The figure illustrates the following basic process:

- The data object is accessed for update (DIVINF).
- The dynamic or extended common is mapped to part of the data object (DIVVWF).
- The window contents are saved into the previously empty data object (DIVSAV).
- The dynamic or extended 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 common, you simply specify the relative position of the area using the *mapnum* parameter on the

DIVVWF subroutine. For example, for the data object shown in Figure 51 on page 362, the value 1 indicates Experiment 1 Data, the value 2 indicates Experiment 2 Data, and so on.

If the 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) = 1.000
    MyArray(256) = 2.000
    MyArray(512) = 3.000

```

Figure 54 (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 54 (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 or extended commons to different parts of the data object, as long as the parts do not overlap. Each common can be mapped to only one part of the data object at a time; if you attempt to map it simultaneously to another part, the first part becomes unmapped, and any changes that you haven't saved are discarded.

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 common and a data object, the varying-view method permits multiple 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 common.

Because the dynamic or extended commons may vary in length, mapping them to the data object also requires more user control than with the fixed-view method because you are responsible for 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 portion 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 or extended 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 55 and Figure 56 on page 379 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 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 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 55 (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 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 commons.

    Call WORK

C Save changed data in the 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 55 (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 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 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 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 56 (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 commons.

      Call WORK

C Save changed data in the 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 56 (Part 2 of 2). Example 3 of Data-in-Virtual Subroutines: Varying-View Method

Remapping a Common to Different Parts of the Data Object

With either the fixed-view method or the varying-view method, you can remap a dynamic or extended common to view and update different parts of the data object. In other words, you can map the common to one part of the data object, process the data and save your changes, then remap the common to another part of the data object to process more data.

In the fixed-view example shown in Figure 57, 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 57. Example of Remapping: Fixed-View Method

Resetting a 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 or extended 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 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 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 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 and extended commons are allocated in units of pages (4096 bytes per page), using 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 common, or else view each part simultaneously with different commons.

Chapter 11. Extended Error-Handling Topics

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 AFBUEPT) 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 VSF2UEPT, and the macro can 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.

Parallel Considerations: Each parallel task has its own error option table. A parallel thread uses the error option table associated with the parallel task in which it runs. Locks must be used to coordinate the setting and retrieving entries in the error option table within a parallel task.

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 389. 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*4 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 to alter should not be in a CALL statement because they are subject to change there.

Because *data1* and *data2* are the parameters that the error monitor passes to a user-written routine to correct the detected error, take care to ensure 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 provide a user-written exit routine.

Example:

```
CALL ERRMON (MYMSG,ICODE,615,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 615; 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

An INTEGER*4 constant, variable or array element whose value is the error number in the option table. An error message will be printed if any number not within the range of the option table is used.

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

An INTEGER*4 constant, variable or array element whose value is the error number. An error message will be printed if any number not within the range of the option table is used.

inoal

An INTEGER*4 constant, variable or array element which 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*4 constant, variable or array element 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*4 constant, variable or array element whose value can 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 than 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 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 (610,20,5,0,MYERR,620)
```

This example specifies the following:

- Error condition 610 (*ierno*).
- The error condition can occur up to 20 times (*inoaf*).
- The corresponding error message can 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 610 to 620 (*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 60 on page 393 through Figure 62 on page 397.

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 60 on page 393 through Figure 62 on page 397.
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. 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. If this routine was written in 24 bit mode, it can be necessary to switch to 24 bit mode before running the routine, and switch back to 31 bit mode before exiting. 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, since control flow and variable usage information will not be known to the optimizer.

Under CMS and MVS, 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.

In the XA or ESA environment, VS FORTRAN passes control to the user-written exit routine in 31-bit addressing mode. The routine can switch to 24-bit addressing mode before program execution but must switch back to 31-bit addressing mode before returning control to the caller. It also must be able to handle the fact that the save area, parameters, and return address can reside above the 16M line.

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 can take if they want to correct an error are described in Figure 60 on page 393 through Figure 62 on page 397.

ERRSTR Subroutine

The ERRSTR subroutine stores an entry in the option table.

Syntax

CALL ERRSTR (*ierno*, *tabent*)

ierno

An INTEGER*4 constant, variable or array element whose value is 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 error option table entries is shown in Figure 58. Figure 59 on page 390 lists the preset information for each error condition. Figure 59 on page 390 through Figure 62 on page 397 summarize the preset information for standard or user-supplied corrective action. The preset entries that cannot be altered are identified in Figure 59.

Figure 58. Error Option Table Entry

Figure 58. Error Option Table Entry																																										
Field Conts.	Field Length	Default Value(1)	Field Description																																							
Number of error occur- rences	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 can 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):																																							
			<table><tr><th>Bit</th><th>Setting</th><th>Explanation</th></tr><tr><td rowspan="2">0</td><td>0*</td><td>Error condition is not an I/O error.</td></tr><tr><td>1</td><td>Error condition is an I/O error. Occurrences are not to be counted if ERR or IOSTAT parameter is given.</td></tr><tr><td rowspan="2">1</td><td>0</td><td>Table entry cannot be modified. (5)</td></tr><tr><td>1*</td><td>Table entry can be modified.</td></tr><tr><td rowspan="2">2</td><td>0*</td><td>Fewer than 256 errors have occurred.</td></tr><tr><td>1</td><td>More than 256 errors have occurred. (Add 256 to error count field above to determine the number.)</td></tr><tr><td rowspan="2">3(6)</td><td>0*</td><td>Do not print buffer with error messages.</td></tr><tr><td>1</td><td>Print buffer contents.</td></tr><tr><td>4</td><td>0*</td><td>Reserved.</td></tr><tr><td rowspan="2">5</td><td>0*</td><td>Print messages default number of times only.</td></tr><tr><td>1</td><td>Unlimited printing requested; print for every occurrence of error.</td></tr><tr><td rowspan="2">6(7)</td><td>0</td><td>Do not print traceback map.</td></tr><tr><td>1*</td><td>Print traceback map.</td></tr><tr><td>7</td><td>0*</td><td>Reserved.</td></tr></table>	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.
			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.																																							

Notes:

(1) The default values shown apply to all error numbers (including additional user entries) unless excepted by a footnote.

(2) Errors 207, 208, 209, and 215 are set as unlimited, and errors 162, 163, 164, 165, 167, 168, 205, 217, 230, and 240 are set to 1.

(3) An unlimited number of errors can cause the Fortran job to loop.

(4) Errors 162, 163, 164, 165, 167, 168, 230, and 240 are set to 1.

(5) The entry for errors 97, 116, 117, 141, 142, 145, 146, 147, 148, 149, 150, 153, 156, 158, 162, 163, 164, 165, 168, 176, 177, 178, 179, 190, 198, 202, 205, 230, and 240 cannot be modified.

(6) The entry is set to 0 except for errors 212, 215, 218, 221, 222, 223, 224, 225, 227, and 238.

(7) The entry is set to 1 except for error 205.

Figure 59 (Page 1 of 3). Option Table Default Values

Error Code	No. of Errors Allowed	No. of Messages Allowed	I/O Error Counting Suppressed(1)	Modifiable Entry	Print Buffer Content	Trace-Back Allowed	Standard Corrective Action
84	10	5	Yes	Yes	No	Yes	Yes
85	Unlimited	Unlimited	No	Yes	No	No	NA
86	10	5	Yes	Yes	No	Yes	yes
87	Unlimited	Unlimited	No	Yes	No	No	NA
88 - 89	10	5	No	Yes	No	Yes	Yes
90 - 95	10	5	Yes	Yes	No	Yes	Yes
96	10	5	No	Yes	No	Yes	Yes
97	1	1	No	No	No	Yes	No
98 - 99	10	5	No	Yes	No	Yes	Yes
100	10	5	Yes	Yes	No	Yes	Yes
101	10	5	Yes	Yes	No	Yes	Yes
102 - 105	10	5	Yes	Yes	No	Yes	Yes
106 - 109	1	1	Yes	Yes	No	Yes	Yes
110 - 111	10	5	Yes	Yes	No	Yes	Yes
112	Unlimited	5	No	Yes	No	Yes	Yes (2)
113	10	5	Yes	Yes	No	Yes	Yes
114 - 115	1	1	Yes	Yes	No	Yes	Yes
116 - 117	1	1	No	No	No	Yes	No
118 - 119	10	5	No	Yes	No	Yes	Yes
120 - 129	1	1	Yes	Yes	No	Yes	Yes
130 3,4	1	1	Yes	Yes	No	Yes	Yes
131 - 139	1	1	Yes	Yes	No	Yes	Yes
140	10	5	Yes	Yes	No	Yes	Yes
141 - 142	1	1	No	No	No	No	No
143 - 144	Unlimited	5	No	Yes	No	Yes	Yes
145	1	1	No	No	No	No	No
146 - 150	1	1	No	No	No	Yes	No
151	Unlimited	Unlimited	No	Yes	No	No	NA
152	10	5	Yes	Yes	No	Yes	Yes
153	1	1	No	No	No	Yes	No
154	Unlimited	5	No	Yes	No	Yes	Yes
155	10	5	Yes	Yes	No	Yes	Yes
156	1	1	No	No	No	Yes	Yes
157	10	5	No	Yes	No	Yes	Yes
158	1	1	No	No	No	Yes	Yes
159 - 160	10	5	No	Yes	No	Yes	Yes
161	10	5	Yes	Yes	No	Yes	Yes
162 - 165	1	1	Yes	No	No	Yes	Yes
166	10	5	Yes	Yes	No	Yes	Yes
167 (3)	1	1	Yes	Yes	No	Yes	Yes
168	1	1	No	No	No	Yes	Yes

Figure 59 (Page 2 of 3). Option Table Default Values

Error Code	No. of Errors Allowed	No. of Messages Allowed	I/O Error Counting Suppressed(1)	Modifiable Entry	Print Buffer Content	Trace-Back Allowed	Standard Corrective Action
169 - 170	10	5	Yes	Yes	No	Yes	Yes
171	10	5	Yes	Yes	No	Yes	Yes
172 - 175	10	5	Yes	Yes	No	Yes	Yes
176	1	1	No	No	No	Yes	No
177 - 179	1	1	No	No	No	Yes	No
180 - 186	10	5	Yes	Yes	No	Yes	Yes
187 - 189	10	5	No	Yes	No	Yes	Yes
190	Unlimited	5	No	No	No	Yes	Yes
191 - 197	10	1	No	Yes	No	No	No
198	Unlimited	5	No	No	No	Yes	Yes
199	10	5	No	Yes	No	Yes	No
200	10	5	Yes	Yes	No	Yes	No
201	Unlimited	5	No	Yes	No	Yes	Yes
202	1	1	No	No	No	Yes	No
203 - 204	10	5	No	Yes	No	Yes	Yes
205	1	1	No	No	No	No	No
206	10	5	No	Yes	No	Yes	Yes
207 - 208	Unlimited	5	No	Yes	No	Yes	Yes
209	Unlimited	5	No	Yes	No	Yes	Yes (2)
210	10(5)	5	No	Yes	No	Yes	Yes (2)
211	10	5	No	Yes	No	Yes	Yes
212	10	5	No	Yes	Yes	Yes	Yes
213	10	5	No	Yes	No	Yes	Yes
214	10	5	Yes	Yes	No	Yes	Yes
215	Unlimited	5	No	Yes	Yes	Yes	Yes
217 (6)	1	1	NA	Yes	No	Yes	Yes
218	10(7)	5	Yes	Yes	Yes	Yes	Yes
219	10(8)	5	Yes	Yes	No	Yes	Yes
220	10	5	Yes	Yes	No	Yes	Yes
221 - 225	10	5	No	Yes	Yes	Yes	Yes
226	10	5	No	Yes	No	Yes	Yes
227	10	5	No	Yes	Yes	Yes	Yes
228	10	5	No	Yes	No	Yes	Yes
229	Unlimited	Unlimited	No	Yes	No	No	NA
230	1	1	No	No	No	Yes	No
231 - 232	10	5	Yes	Yes	No	Yes	Yes
233	10	5	No	Yes	No	Yes	Yes
234 - 236	10	5	Yes	Yes	No	Yes	Yes
237	10	5	Yes	Yes	No	Yes	Yes
238	10	5	No	Yes	Yes	Yes	Yes
239	10	5	No	Yes	No	Yes	Yes

Figure 59 (Page 3 of 3). Option Table Default Values

Error Code	No. of Errors Allowed	No. of Messages Allowed	I/O Error Counting Suppressed(1)	Modifiable Entry	Print Buffer Content	Trace-Back Allowed	Standard Corrective Action
240	1	1	No	No	No	Yes	No
241 - 285	10	5	No	Yes	No	Yes	Yes
286	10	5	Yes	Yes	No	Yes	Yes
287 - 311	10	5	No	Yes	No	Yes	Yes

Notes:

- (1) A YES in this column indicates that the error condition is an I/O error. Occurrences of these errors are not to be counted if the ERR or IOSTAT parameter has been specified in the I/O statement. The ERR and IOSTAT parameters are honored only for those codes whose default value in this column is YES and for the following error codes that can be enabled with the CNVIOERR option on READ statements and formatted WRITE statements: 206, 212, 213, 215, 225, 226 and 238.
Even if the default value YES is changed to NO for a given error code during customization, ERR and IOSTAT are still honored.
- (2) No corrective action is taken except to continue execution. For boundary alignment, the corrective action is part of the support for misalignment. For errors 112 and 209, no user corrective action can be specified.
- (3) For error codes 130 and 167, VSAM error information is returned in IOSTAT as two consecutive INTEGER*2 values. The first is the VSAM return code (the value in register 15 after the execution of the VSAM macro). The second is the VSAM error or feedback code.
- (4) If an error code of 130 results from processing an OPEN statement, a return code of 4 from the VSAM OPEN macro is not considered an error. The ERR=label will not be branched to, but IOSTAT is still set to the appropriate values.
- (5) Ten errors are allowed only if a user error routine specifies that user-corrective action should be taken. In all other cases, message AFB240I is printed after one occurrence of the error, and the program is terminated immediately.
- (6) The end-of-data set condition is not considered an error if the END parameter is present in a READ statement. No message or traceback is printed and the error count is not altered.
- (7) For an input/output error, the buffer can have been partially filled or not filled at all when the error was detected. Thus, the buffer contents could be blank when printed. When an ERR parameter is specified in a READ statement, it is honored even though the error occurrence is greater than the amount allowed.
- (8) Because a single WRITE performed in a loop could cause 10 occurrences of the message for the same missing DD statement, the count field does not necessarily mean that up to 10 missing DD statements will be detected in a single debugging run.

For user-supplied corrective action, see Figure 60 on page 393.

Message Corrective Action Cross-Reference Tables

Figure 60 (Page 1 of 3). Corrective Action after Error

Error Code	Parameters Passed To User Exit	Standard Corrective Action	User-Supplied Corrective Action
120	A,B,D	I/O statement ignored.	Note 1
122-123	A,B,D	I/O statement ignored.	Note 1
124	A,B,D,L	I/O statement ignored.	Note 1
125	A,B,D,J,K,L	I/O statement ignored.	Note 1
126	A,B,D,J,K,L	I/O statement ignored. If the key argument was supplied on a KEY parameter, file position is lost. If the key argument was supplied on a KEYGE or KEYGT parameter, the file is positioned following the last record in the file.	The condition described by error code 126 is not considered an error if the NOTFOUND parameter is specified on the READ statement. See Note 1.
127, 128	A,B,D	I/O statement ignored.	Note 1
129	A,B,D,M,N	The record is not written or rewritten. File position is lost.	Note 1
130	A,B,D,O,P	The I/O statement is not processed further. File position is lost.	Note 1
131-134	A,B,D	The file was not successfully opened for use by the program even though the VSAM KSDS was internally opened for access via VSAM.	Note 1
135	A,B,D,M,N	File position is lost.	The condition described by error code 135 is not considered an error if the DUPKEY parameter is specified on the WRITE or REWRITE statement. See Note 1.
136, 137	A,B,D	I/O statement ignored.	Note 1
138	A,B,D	The file was not successfully opened for use by the program even though the VSAM KSDS was internally opened for access via VSAM.	Note 1
139	A,B,D,M, N,Q	The record is not rewritten. File position is lost.	Note 1
140	A,B,D,M,N	The I/O statement is not processed further. The file remains available for the loading of subsequent records.	Note 1
206	A,B,I	I=low order part of number for input too large.	User can alter I. Note 2
211	A,B,C	Treat format field containing C as end of FORMAT statement.	If compiled FORMAT statement, put hexadecimal equivalent of character in C. If variable format, move EBCDIC character into C. See Note 3.
212	A,B,D	<i>Input:</i> Ignore remainder of I/O list. <i>Output:</i> Continue by starting new output record, if there are more entries on the I/O list. Supply carriage control character if required by Option Table.	Note 1
213	A,B,D	Ignore remainder of I/O list.	Note 1

Figure 60 (Page 2 of 3). Corrective Action after Error

Error Code	Parameters Passed To User Exit	Standard Corrective Action	User-Supplied Corrective Action
214	A,B,D	<i>Input:</i> Ignore remainder of I/O list. Ignore input/output request if for ASCII tape. <i>Output:</i> 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 cannot 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 can 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 60 (Page 3 of 3). Corrective Action after Error

Error Code	Parameters Passed To User Exit	Standard Corrective Action	User-Supplied Corrective Action
311	A,B,I8	I=low order part of number for input too large.	User can alter I8 Note 6

Notes:

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)
I8	Result after conversion (INTEGER*8)
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)

- 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, cannot 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.
- The user exit routine can supply an alternative answer for the setting of the result value. The routine should always set an INTEGER*4 variable and the VS FORTRAN Version 2 library will load either a fullword or halfword depending on the length of the argument causing the error.
- Alternatively, the user can set the return code to 0, thus requesting a standard corrective action.
- 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 438.
- If LANGLVL(66), then LOGICAL*1.
 - If LANGLVL(77), then CHARACTER*1.
- The user exit routine can supply an alternative answer for the setting of the result value. The routine should always set an INTEGER*8 variable and the VS FORTRAN Version 2 library will use the value to replace the item in error.

Figure 61. Corrective Action after Program Interrupt

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 can alter D(3).
208	A,B,D,I	Underflow Exponent underflow (Interrupt code D)	The result register is set to zero.	The user can 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 can 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)

Notes:

(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 can supply an alternative answer for the setting of the result register. This is accomplished by replacing the value in D. Although the interrupt can 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 can 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 can be specified as:

```
SUBROUTINE MYEXIT(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.

Figure 62 (Page 1 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	Fortran Reference ¹	Invalid Argument Range	Standard Corrective Action ^{2,3}	Options Parameters Passed to User Exit ⁴
118	XA = X**Y	$X < 0, Y \neq 0$	$XA = X ^{**Y}$	A, B, X, Y
119	DA = D**DB	$D < 0, DB \neq 0$	$DA = D ^{**DB}$	A, B, D, DB
241	IA = I**J	$I = 0, J \leq 0$	$IA = 0$	A, B, I, J
242(5)	XA = X**I	$X = 0, I = 0$ $X = 0, I < 0$	$XA = 1$ $XA = \Omega$	A, B, X, I
243(5)	DA = D**I	$D = 0, I = 0$ $D = 0, I < 0$	$DA = 1$ $DA = \Omega$	A, B, D, I
244	XA = X**Y	$X = 0, Y = 0$ $X = 0, Y < 0$	$XA = 1$ $XA = \Omega$	A, B, X, Y
245	DA = D**DB	$D = 0, DB = 0$ $D = 0, DB < 0$	$DA = 1$ $DA = \Omega$	A, B, D, DB
246	CA = C**I	$C = 0+0i, I = 0$ $C = 0+0i, I < 0$	$CA = 1+0i$ $CA = \Omega+0i$	A, B, C, I
247	CDA = CD**I	$C = 0+0i, I = 0$ $C = 0+0i, I < 0$	$CDA = 1+0i$ $CDA = \Omega+0i$	A, B, CD, I
248(5)	QA = Q**I	$Q = 0, I = 0$ $Q = 0, I < 0$	$QA = 1$ $QA = \Omega$	A, B, Q, I
249	QA = Q**QB	$Q = 0, QB = 0$ $Q = 0, QB < 0$ $Q < 0, QB \neq 0$	$QA = 1$ $QA = \Omega$ $QA = Q ^{**QB}$	A, B, Q, QB
250	QA = Q**QB	$\log \text{sub2} (Q) * QB \geq 252$	$QA = \Omega$	A, B, Q, QB
251	XA = SQRT(X)	$X < 0$	$XA = X ^{*(1/2)}$	A, B, X
252	XA = EXP(X)	$X > 174.673$	$XA = \Omega$	A, B, X
253	XA = ALOG(X)	$X = 0$ $X < 0$	$XA = -\Omega$ $XA = \log X $	A, B, X
	XA = ALOG10(X)	$X = 0$ $X < 0$	$XA = -\Omega$ $XA = \log \text{sub10} X $	A, B, X
254	XA = COS (X) XA = SIN (X)	$ X \geq (2 \text{ sup } 18) \pi$	$XA = 1 / \text{sqrt } 2$	A, B, X
255	XA = ATAN2 (X,Y)	$X = 0, Y = 0$	$XA = 0$	A, B, X, Y
256	XA = SINH (X) XA = COSH (X)	$ X \geq 175.366$	$XA = (\text{sign of } X) \Omega$ $XA = \Omega$	A, B, X
257	XA = ASIN (X)	$X > 1.0$ $X < -1.0$	$XA = \pi / 2$ $XA = -\pi / 2$	A, B, X
	XA = ACOS (X)	$X > 1.0$ $X < -1.0$	$XA = 0$ $XA = \pi$	A, B, X
258	XA = TAN (X) XA = COTAN (X)	$ X \geq (2 \text{ sup } 18) \pi$	$XA = 1$	A, B, X
259	XA = TAN (X)	X is too close to an odd multiple of $\pi / 2$	$XA = \Omega$	A, B, X
	XA = COTAN (X)	X is too close to a multiple of π	$XA = \Omega$	A, B, X
260	QA = 2**Q	$Q \geq 252$	$QA = \Omega$	A, B, Q
261	DA = DSQRT (D)	$D < 0$	$DA = D ^{*(1/2)}$	A, B, D
262	DA = DEXP (D)	$D > 174.673$	$DA = \Omega$	A, B, D

Figure 62 (Page 2 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	Fortran Reference ¹	Invalid Argument Range	Standard Corrective Action ^{2,3}	Options Parameters Passed to User Exit ⁴
263	DA = DLOG (D)	D = 0 D < 0	DA = -Ω DA = log D	A, B, D
	DA = DLOG10 (D)	D = 0 D < 0	DA = -Ω DA = log sub10 D	A, B, D
264	DA = DSIN (D) DA = DCOS (D)	D ≥ (2 sup 50) π	DA = 1 / sqrt 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 D) Ω DA = Ω	A, B, D
267	DA = DASIN (D)	D > 1.0 D < -1.0	DA = π / 2 DA = - π / 2	A, B, D
	DA = DACOS (D)	D > 1.0 D < -1.0	DA = 0 DA = π	A, B, D
268	DA = DTAN (D) DA = DCOTAN (D)	D ≥ (2 sup 50) π	DA = 1	A, B, D
269	DA = DTAN (D)	D is too close to an odd multiple of π / 2	DA = Ω	A, B, D
	DA = DCOTAN (D)	D is too close to a multiple of π	DA = Ω	A, B, D
270	CQA = CQ**I	CQ = 0+0i, I = 0 CQ = 0+0i, I < 0	CQA = 1+0i CQA = Ω+0i	A, B, CQ, I
271(6)	Z = CEXP (C)	Xsub1 > 174.673	Z = Ω(cos Xsub2 + i sin Xsub2)	A, B, C
272(6)	Z = CEXP (C)	X sub2 ≥ (2**18) π	Z = e **(Xsub1) +0i	A, B, C
273(6)	Z = CLOG (C)	C = 0+0i	Z = - Ω+0i	A, B, C
274(6)	Z = CSIN (C) Z = CCOS (C)	Xsub1 ≥ (2**18)π	Z = 0+i sinh Xsub2 Z = cosh(Xsub2)+0i	A, B, C
275(6)	Z = CSIN (C) Z = CCOS (C) Z = CSIN (C) Z = CCOS (C)	Xsub2 > 174.673 Xsub2 < -174.673	Z = Ω (sin(Xsub1) + i cos(Xsub1))/2 Z = Ω (cos(Xsub1) - i sin(Xsub1))/2 Z = Ω (sin(Xsub1) - i cos(Xsub1))/2 Z = Ω (cos Xsub1 + i sin Xsub1)/2	A, B, C
276(7)	Z = CQEXP (CQ)	Xsub1 > 174.673	Z = Ω (cos Xsub2 + i sin Xsub2)	A, B, CQ
277(7)	Z = CQEXP (CQ)	Xsub2 > 2**100	Z = e**(Xsub1) + 0i	A, B, CQ
278(7)	Z = CQLOG (CQ)	CQ = 0+0i	Z = - Ω+0i	A, B, CQ
279(7)	Z = CQSIN (CQ) Z = CQCOS (CQ)	Xsub1 ≥ 2**100	Z = 0+i sinh Xsub2 Z = cosh Xsub2+0i	A, B, CQ
280(7)	Z = CQSIN (CQ) Z = CQCOS (CQ) Z = CQSIN (CQ) Z = CQCOS (CQ)	Xsub2 > 174.673 Xsub2 < -174.673	Z = Ω (sin Xsub1 + i cos Xsub1)/2 Z = Ω (cos Xsub1 - i sin Xsub1)/2 Z = Ω (sin Xsub1 - i cos Xsub1)/2 Z = Ω (cos Xsub1 + i sin Xsub1)/2	A, B, CQ

Figure 62 (Page 3 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	Fortran Reference ¹	Invalid Argument Range	Standard Corrective Action ^{2,3}	Options Parameters Passed to User Exit ⁴
281(8)	Z = CDEXP (CD)	Xsub1 > 174.673	$Z = \Omega (\cos X_{\text{sub}2} + i \sin X_{\text{sub}2})$	A, B, CD
282(8)	Z = CDEXP (CD)	$ X_{\text{sub}2} \geq (2^{**}50) \pi$	$Z = e^{**}(X_{\text{sub}1})+0i$	A, B, CD
283(8)	Z = CDLOG (CD)	CD = 0+0i	$Z = -\Omega+0i$	A, B, CD
284(8)	Z = CDSIN (CD) Z = CDCOS (CD)	$ X_{\text{sub}1} \geq (2^{**}50) \pi$	$Z = 0+i \sinh X_{\text{sub}2}$ $Z = \cosh X_{\text{sub}2}+0i$	A, B, CD
285(8)	Z = CDSIN (CD) Z = CDCOS (CD) Z = CDSIN (CD) Z = CDCOS (CD)	Xsub2 > 174.673 Xsub2 < -174.673	$Z = \Omega(\sin X_{\text{sub}1} + i \cos X_{\text{sub}1})/2$ $Z = \Omega(\cos X_{\text{sub}1} - i \sin X_{\text{sub}1})/2$ $Z = \Omega(\sin X_{\text{sub}1} - i \cos X_{\text{sub}1})/2$ $Z = \Omega(\cos X_{\text{sub}1} + i \sin X_{\text{sub}1})/2$	A, B, CD
289	QA = QSQRT (Q)	Q < 0	$QA = Q ^{**}(1/2)$	A, B, Q
290	XA = GAMMA (X)	$X \leq 2^{**}(-252)$ or $X \geq 57.5744$	$XA = \Omega$	A, B, X
291	XA = ALGAMA (X)	$X \leq 0$ or $X \geq 4.2937 * 10^{**}73$	$XA = \Omega$	A, B, X
292	QA = QEXP (Q)	Q > 174.673	$QA = \Omega$	A, B, Q
293	QA = QLOG (Q)	Q = 0 Q < 0	$QA = -\Omega$ $QA = \log Q $	A, B, Q
	QA = QLOG10 (Q)	Q = 0 Q < 0	$QA = -\Omega$ $QA = \log_{\text{sub}10} Q $	A, B, Q
294	QA = QSIN (Q) QA = QCOS (Q)	$ Q \geq 2^{**}100$	$QA = 1 / \sqrt{2}$	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)	$ Q \geq 175.366$	$QA = (\text{sign of } Q) \Omega$ $QA = \Omega$	A, B, Q
297	QA = QARSIN (Q)	Q > 1.0 Q < -1.0.	$QA = \pi / 2$ $QA = -\pi / 2$	A, B, Q
	QA = QARCOS (Q)	Q > 1.0 Q < -1.0	QA = 0 QA = π	A, B, Q
298	QA = QTAN (Q) QA = QCOTAN (Q)	$ Q \geq 2^{**}100$	QA = 1	A, B, Q
299	QA = QTAN (Q)	Q is too close to an odd multiple of $\pi / 2$	QA = Ω	A, B, Q
	QA = QCOTAN (Q)	Q is too close to a multiple of π	QA = Ω	A, B, Q
300	DA = DGAMMA (D)	$D \leq 2^{**}(-252)$ or $D \geq 57.5774$	DA = Ω	A, B, D
301	DA = DLGAMA (D)	$D \leq 0$ or $D \geq 4.2937 * 10^{**}73$	DA = Ω	A, B, D
302	I8 = L**J	L = 0, J ≤ 0	I8 = 0	A, B, L, J
303	IA = I**K	I = 0., K ≤ 0	IA = 0	A, B, I, K
304	I8 = L**K	L = 0, K ≤ 0	I8 = 0	A, B, L, K
305(5)	XA = X**L	X = 0, L = 0 X = 0, L < 0	XA = 1 XA = Ω	A, B, X, L

Figure 62 (Page 4 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	Fortran Reference ¹	Invalid Argument Range	Standard Corrective Action ^{2,3}	Options Parameters Passed to User Exit ⁴
306(5)	DA = D**L	D = 0, L = 0 D = 0, L < 0	DA = 1 DA = Ω	A, B, D, L
307(5)	QA = Q**L	Q = 0, L = 0 Q = 0, L < 0	QA = 1 QA = Ω	A, B, Q, L
308	CA = C**L	C = 0+0i, L = 0 C = 0+0i, L < 0	CA = 1+0i CA = Ω +0i	A, B, C, L
309	CDA = CD**L	C = 0+0i, L = 0 C = 0+0i, L < 0	CDA = 1+0i CDA = Ω +0i	A, B, CD, L
310	CQA = CQ**L	CQ = 0+0i, L = 0 CQ = 0+0i, L < 0	CQA = 1+0i CQA = Ω +0i	A, B, CQ, L

Notes:

- (1) The variable types are as follows:

Variable (Type) Meaning

A (INTEGER*4) Address of return code field.

B (INTEGER*4) The error number.

I,IA,J (INTEGER*4)

L,I8,K (INTEGER*8)

X,XA,Y (REAL*4)

D,DA,DB (REAL*8)

Q,QA,QB (REAL*16)

C,CA (COMPLEX*8)

CD,CDA (COMPLEX*16)

CQ,CQA (COMPLEX*32)

Z A complex variable which has the length of the function argument when it appears.

Xsub1,Xsub2 Xsub1 and Xsub2 are real and imaginary parts, respectively, of C, CD, or CQ.

- (2) $\Omega = (16^{**63})(1 - 16^{**(-6)})$ for regular precision routines, $\Omega = (16^{**63})(1 - 16^{**(-14)})$ for double precision routines, and $\Omega = (16^{**63})(1 - 16^{**(-28)})$ for extended precision routines.
- (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 $\text{BASE}^{**\text{exponent}}$ and $\text{BASE}^{**(-\text{exponent})}$ are representable.
- (6) For errors 271 through 275, $C = \text{Xsub1} + i\text{Xsub2}$
- (7) For errors 276 through 280, $\text{CQ} = \text{Xsub1} + i\text{Xsub2}$
- (8) For errors 281 through 285, $\text{CD} = \text{Xsub1} + i\text{Xsub2}$

Chapter 12. 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 operating system.

The multitasking facility subroutines cannot be used in a parallel program. All references in this chapter to parallel subroutines, scheduled parallel subroutines, and main task programs are in MTF terms. For more information on coding parallel subroutines for multitasking, see the *VS FORTRAN Version 2 Programming Guide*.

Note: The multitasking facility subroutines cannot be used to reference data in an extended common block. For more information on extended common, see the *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.

The NTASKS subroutine returns a value of zero when the AUTOTASK keyword is not in effect.

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 in effect.
1 - 99	The number of subtasks specified with the AUTOTASK keyword.

Notes:

1. NTASKS can 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 can 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 can 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 (subname [, arg1 [, arg2][...]])
```

subname

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, *subname* 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, *subname* is a character expression.

For Language Level 77, the character expression *subname* can 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, *subname* 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 can 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 can 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 can 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 an MVS operating system 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 can 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 can 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 can 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. Summary of the VS FORTRAN Version 2 Language

VS FORTRAN Version 2 is designed according to the specifications of the following external standards:

- *American National Standard Programming Language FORTRAN, ANSI X3.9-1978, and International Organization for Standardization ISO 1539-1980 Programming Languages—FORTRAN.* These two standards are technically equivalent, and are referred to informally as the 77 level.
- *American Standard FORTRAN, X3.9-1966, and International Organization for Standardization ISO R 1539-1972 Programming Languages—FORTRAN.* These two standards are technically equivalent, and are referred to informally as the 66 level.

Beginning with Release 3, VS FORTRAN Version 2 supports the SAA Common Programming Interface FORTRAN definition. VS FORTRAN Version 2 not only supports SAA and both of the above standards, but also includes many IBM language extensions.

Source Language Flaggers

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 compile-time option specifies whether this flagging is to be performed.

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 compile-time option specifies whether this flagging is to be performed.

For more information about the compile-time options, see the *VS FORTRAN Version 2 Programming Guide*.

Major Elements of the VS FORTRAN Version 2 Language

Figure 63 on page 406 lists the major elements of the VS FORTRAN Version 2 language. The table shows whether:

- An element is supported at the 77 and the 66 levels
- An element is standard (Std) or an IBM extension (Ext) to either the 77 or 66 standard
- An element is included in the Systems Application Architecture FORTRAN definition
- An element is flagged for not conforming to the FORTRAN 77 standard (FIPS), if "No" or "Yes - Ext" appears in the corresponding column
- An element is flagged for not conforming to the Systems Application Architecture FORTRAN definition (SAA), if "No" appears in the corresponding column.

Figure 63 (Page 1 of 5). Major Elements of the VS FORTRAN Version 2 Language

Statement or Feature	Use	In LANTLRVL (77)?	In LANTLRVL (66)?	In SAA?
Ampersand (&)	Allowed as special character	No	Yes - Ext	No
ALLOCATE	Allocate storage	Yes - Ext	No	No
ASSIGN	Assigns GOTO targets	Yes - Std	Yes - Std	Yes
	Assigns FORMAT labels	Yes - Std	No	Yes
Assignment statements	Assign values to arithmetic and logical data items	Yes - Std	Yes - Std	Yes
	Assign values to character data items Assign values to unsigned data items	Yes - Std Yes - Ext	No	Yes
Asynchronous I/O	Read/write in asynchronous mode	Yes - Ext	Yes - Ext	No
AT	Specifies beginning of debugging packet	Yes - Ext	Yes - Ext	No
AUTOMATIC	Controls storage class	Yes - Ext	No	No
BACKSPACE	Repositions file at previous record	Yes - Std	Yes - Std	Yes
Binary constants	For binary data values	Yes - Ext	No	No
BLOCK DATA	Identifies a data subprogram	Yes - Std	Yes - Std	Yes
CALL	Transfers control to a subroutine	Yes - Std	Yes - Std	Yes
Character data type	Allows character (string) data	Yes - Std	No	Yes
CLOSE	Disconnects file from a program	Yes - Std	No	Yes
Columns 1 to 5	Can be nonblank on continuation line	Yes - Ext	Yes - Ext	No
COMMON	Defines storage shared between programs	Yes - Std	Yes - Std	Yes
	Allows both character and noncharacter data in one block	Yes - Std	No	Yes
Complex data type	Complex numbers of single precision	Yes - Std	Yes - Std	Yes
	Complex numbers of double and extended precision	Yes - Ext	Yes - Ext	Yes
CONTINUE	Nonoperational executable statement for programming convenience	Yes - Std	Yes - Std	Yes
Currency symbol (\$)	Can be used in names	Yes - Ext	Yes - Ext	No
DEALLOCATE	Deallocate storage	Yes - Ext	No	No
DATA	Initializes variables and array elements	Yes - Std	Yes - Std	Yes
	Initializes variables and arrays with implied DO loops if desired	Yes - Std	No	Yes
DEBUG and END DEBUG	Delimit the debugging packet portion of a program	Yes - Ext	Yes - Ext	No
DEFINE FILE	Specifies a direct-access file	No	Yes - Ext	No
DELETE	Deletes record from a KSDS file	Yes - Ext	No	No
DIMENSION	Defines arrays of up to three dimensions	Yes - Std	Yes - Std	Yes
	Defines arrays of up to seven dimensions	Yes - Std	Yes - Ext	Yes
	Defines arrays with adjustable size	Yes - Std	Yes - Std	Yes
	Defines arrays with explicit lower bounds (which can be positive or negative)	Yes - Ext	No	Yes
Direct-access I/O	Read/write by record number	Yes - Std	Yes - Ext	Yes

Figure 63 (Page 2 of 5). Major Elements of the VS FORTRAN Version 2 Language

Statement or Feature	Use	In LANTLR (77)?	In LANTLR (66)?	In SAA?
DISPLAY	Displays data within a debugging packet	Yes - Ext	Yes - Ext	No
DO	Gives a convenient way to program loops (using integer DO variables)	Yes - Std	Yes - Std	Yes
	Real and double precision DO variables are allowed; negative incrementation parameter is allowed	Yes - Std	No	Yes
DOAFTER	Identifies a block of statements that each virtual processor must process once after running the iterations of the loop.	Yes - Ext	Yes - Ext	No
DOBEFORE	Identifies a block of statements that each virtual processor must process once before running the iterations of the loop.	Yes - Ext	Yes - Ext	No
DOEVERY	Identifies a block of statements that make up the body of the loop.	Yes - Ext	Yes - Ext	No
DO WHILE	Initiates processing of program loops based on evaluation of a logical expression	Yes - Ext	No	No
EJECT	Starts new page of source listing	Yes - Ext	Yes	No
END	Marks end of program unit	Yes - Std	Yes - Std	Yes
	Terminates program processing	Yes - Std	No	Yes
END DO	Terminates processing of a DO , PARALLEL DO, or DO WHILE loop	Yes - Ext	No	No
end of line commentary	The "!" indicates the beginning of a comment	Yes - Ext	Yes - Ext	No
ENDFILE	Writes end-of-file record	Yes - Std	Yes - Std	Yes
END SECTIONS	Indicates the end of a group of sections	Yes - Ext	Yes - Ext	No
ENTRY	Specifies alternate entry points into subprograms	Yes - Std	Yes - Std	Yes
EQUIVALENCE	Defines shared storage	Yes - Std	Yes - Ext	Yes
	Can relate character and noncharacter data	Yes - Std	No	Yes
EXIT	Stops the running of a parallel loop	Yes - Ext	Yes - Ext	No
Explicit type statements	Define data types of specific variables	Yes - Std	Yes - Std	Yes
Expressions	Manipulate arithmetic, relational, or logical items, or other expressions	Yes - Std	Yes - Std	Yes
	Manipulate character items or arithmetic double precision or complex items	Yes - Std	No	Yes
EXTERNAL	Defines linked subprograms	Yes - Std	Yes - Std	Yes
FIND	Locates next input record	No	Yes - Ext	No
FORMAT	Defines record formats	Yes - Std	Yes - Std	Yes
	Character constants and run-time formats allowed	Yes - Std	No	Yes
Free-form source	Relaxes format rules for source program	Yes - Ext	Yes - Ext	No
FUNCTION	Identifies a function subprogram	Yes - Std	Yes - Std	Yes
GENERIC	Allows automatic function selection	No	Yes - Ext	No
GO TO	Specifies transfers of control	Yes - Std	Yes - Std	Yes
Hexadecimal constants	For hex data values	Yes - Ext	Yes - Ext	No

Figure 63 (Page 3 of 5). Major Elements of the VS FORTRAN Version 2 Language

Statement or Feature	Use	In LANTLRVL (77)?	In LANTLRVL (66)?	In SAA?
Hollerith constants	For initializing variables	Yes - Ext	Yes - Std	No
	As arguments	Yes - Ext	Yes - Ext	No
	As character constants in FORMAT statements	Yes - Ext	Yes - Std	Yes
IF	Specifies alternate paths of processing, using arithmetic and logical IF versions	Yes - Std	Yes - Std	Yes
	Block IF version, using ELSE, ELSE IF, and END IF	Yes - Std	No	Yes
IMPLICIT	Types groups of variables	Yes - Std	Yes - Ext	Yes
	Classifies groups of variables for storage class	Yes - Ext	Yes - Ext	No
INCLUDE	Copies prewritten source statements into program	Yes - Ext	Yes - Ext	Yes
INQUIRE	Retrieves information about a file	Yes - Std	No	Yes
Integer data type	For integer numbers	Yes - Std	Yes - Std	Yes
	Integer*1 (1 byte)	Yes - Ext	Yes - Ext	No
	Integer*2 (2 byte)	Yes - Ext	Yes - Ext	Yes
	Integer or Integer*4 (4 byte)	Yes - Std	Yes - Std	Yes
	Integer*8 (8 byte)	Yes - Ext	Yes - Ext	No
Internal files	Allow easy data conversion	Yes - Std	No	Yes
Intrinsic functions	Supply arithmetic and generic functions	Yes - Std	Yes - Std	Yes
	Supply character and bit functions	Yes - Std	No	Yes
INTRINSIC	Explicitly defines intrinsic functions	Yes - Std	No	Yes
I/O status indicator	Determine success of input/output statement	Yes - Std	No	Yes
Keyed I/O	Read/write by record key value	Yes - Ext	No	No
Length fields	Optional specification for data types	Yes - Ext	Yes - Ext	Yes
List-directed I/O	Read/write formatted data without FORMAT statement	Yes - Std	Yes - Ext	Yes
'Literal constants'	Literal constants enclosed in apostrophes	Yes - Std	Yes - Ext	Yes
LOCAL	Specifies variables, instances of which are provided to each virtual processor	Yes - Ext	Yes - Ext	No
Logical data type	For true/false value	Yes - Std	Yes - Std	Yes
	Logical*1 (1 byte)	Yes - Ext	Yes - Ext	No
	Logical*2 (2 byte)	Yes - Ext	Yes - Ext	No
	Logical or Logical*4 (4 byte)	Yes - Std	Yes - Std	Yes
	Logical*8 (8 byte)	Yes - Ext	Yes - Ext	No
Mixed-mode expressions	Allow mixing of data types	Yes - Std	Yes - Ext	Yes
NAMelist	Read/write referencing named list	Yes - Ext	Yes - Ext	No
NULLIFY	Nullifies a pointer variable	Yes - Ext	No	No
Octal constants	For octal data values	Yes - Ext	No	No
OPEN	Connects files to a program; error routines can be specified	Yes - Std	No	Yes

Figure 63 (Page 4 of 5). Major Elements of the VS FORTRAN Version 2 Language

Statement or Feature	Use	In LANTLR (77)?	In LANTLR (66)?	In SAA?
ORIGINATE	Creates a new parallel task	Yes - Ext	Yes - Ext	No
PARALLEL CALL	Specifies a subroutine to run in parallel	Yes - Ext	Yes - Ext	No
PARALLEL DO	Begins a parallel loop	Yes - Ext	Yes - Ext	No
PARALLEL SECTIONS	Begins a group of parallel sections	Yes - Ext	Yes - Ext	No
PARAMETER	Establishes names for constants	Yes - Std	No	Yes
PAUSE	Suspends program processing temporarily	Yes - Std	Yes - Std	Yes
POINTER	Defines a pointer variable	Yes - Ext	No	No
PRINT	Installation- dependent write statement	Yes - Std	Yes - Ext	Yes
PROGRAM	Names a main program	Yes - Std	No	Yes
PUNCH	Installation- dependent write statement	No	Yes - Ext	No
Quotation mark	Double quote (") allowed as special character	Yes - Ext	No	No
READ	Reads a record from a file	Yes - Std	Yes - Std	Yes
Real data type	Single precision floating-point numbers	Yes - Std	Yes - Std	Yes
	Double precision floating-point numbers	Yes - Std	Yes - Std	Yes
	Extended precision floating-point numbers	Yes - Ext	Yes - Ext	No
Real subscripts	Expressions with floating-point numbers can be used as subscripts	Yes - Ext	Yes - Ext	No
Relational Operators	Compare two arithmetic expressions	Yes - Std	Yes - Std	Yes - Std
Relational Operator Extensions	Compare two arithmetic expressions	Yes - Ext	Yes - Ext	No
RETURN	Returns control to a calling program	Yes - Std	Yes - Std	Yes
REWIND	Repositions to beginning of file	Yes - Std	Yes - Std	Yes
REWRITE	Rewrites record in a KSDS file	Yes - Ext	No	No
SAVE	Saves values after a called program completes executing	Yes - Std	No	Yes
SCHEDULE	Assigns a subroutine to an originated task	Yes - Ext	Yes - Ext	No
SECTION	Indicates the beginning of a group of statements to be processed as a parallel thread	Yes - Ext	Yes - Ext	No
Sequential I/O	Read/write sequential files	Yes - Std	Yes - Std	Yes
Statement functions	Allow convenient programming of expressions	Yes - Std	Yes - Std	Yes
STATIC	Controls storage class	Yes - Ext	No	No
STOP	Terminates program processing	Yes - Std	Yes - Std	Yes
SUBROUTINE	Identifies a subroutine subprogram	Yes - Std	Yes - Std	Yes
Symbolic names	May be 31 characters long	Yes - Ext	No	Yes
TERMINATE	Deletes an originated task	Yes - Ext	Yes - Ext	No
TRACE ON/OFF	Traces specific portions of a program	Yes - Ext	Yes - Ext	No
Underscore character (_)	Can be used in names	Yes - Ext	Yes - Ext	Yes
Unsigned data type	Allows natural numbers	Yes - Ext	No	No
VSAM I/O	Supports ESDS, RRDS, and KSDS files	Yes - Ext	No	No
WAIT FOR ALL CALLS	Causes the program to wait for all subroutines to finish processing before continuing	Yes - Ext	Yes - Ext	No
WAIT FOR ALL TASKS	Waits for all parallel tasks to complete assigned work	Yes - Ext	Yes - Ext	No

Figure 63 (Page 5 of 5). Major Elements of the VS FORTRAN Version 2 Language

Statement or Feature	Use	In L ^{ANGLVL} (77)?	In L ^{ANGLVL} (66)?	In SAA?
WAIT FOR ANY TASK	Waits for any parallel task to complete assigned work	Yes - Ext	Yes - Ext	No
WAIT FOR TASK	Waits for completion of assigned work in a specific parallel task	Yes - Ext	Yes - Ext	No
WRITE	Writes a record into a file	Yes - Std	Yes - Std	Yes

FORTRAN 66 Features Not in FORTRAN 77

LANGLVL(66) instructs the compiler to compile a program according to the 1966 FORTRAN language. Listed here are the features of FORTRAN 66 that are *not* in FORTRAN 77. See Figure 63 on page 406 for FORTRAN 77 features that are not in FORTRAN 66.

- Character constants can be assigned to integer, real, complex, or logical in a DATA statement.
- The ampersand (&) is included in the character set.
- The ampersand (&) must be used instead of the asterisk (*) for an alternate return.
- A program name can be specified only as a compile-time option.
- Arguments are received by value.
- Dummy arguments can be enclosed in slashes to ensure they are received by location.
- DARCOS, DARSIN, ARCOS, and ARSIN used as function names are recognized as functions supplied by VS FORTRAN; DACOS, DASIN, ACOS, and ASIN are recognized as user-supplied function names.
- DEFINE FILE *a* (*m*, *r*, *f*, *v*) [*,a* (*m*, *r*, *f*, *v*)].
- Loops are always executed at least once. (DO statement and implied DO in I/O.)
- EQUIVALENCE statement. (Accept a multidimensional array with one subscript.)
- EXTERNAL statement: If a function name supplied by VS FORTRAN appears in an EXTERNAL statement preceded by an ampersand (&), it is considered a user-supplied function name. If it is not preceded by an ampersand (&), it is considered a function name supplied by VS FORTRAN, except as described below. The following names are always considered user-supplied function names if they appear in an EXTERNAL statement whether or not preceded by an ampersand (&):

ABS, AIMAG, AINT, AMAX0, AMAX1, AMIN0, AMIN1, AMOD, CMPLX, CONJG, DABS, DBLE, DBLEQ, DCMPLX, DCONJG, DDIM, DFLOAT, DIM, DIMAG, DINT, DMAX1, DMIN1, DMOD, DREAL, DSIGN, FLOAT, HFIX, IABS, IDIM, IDINT, IFIX, IMAG, INT, IQINT, ISIGN, MAX, MAX0, MAX1, MIN, MIN0, MIN1, MOD, QABS, QCMPLX, QCONJG, QDIM, QEXT, QEXTD, QFLOAT,

QIMAG, QINT, QMAX1, QMIN1, QMOD, QREAL, QSIGN, REAL, SIGN, SNGL, SNGLQ.

- FIND (*a' r*).
- Function names: ANINT, CHAR, DPROD, DNINT, ICHAR, IDNINT, INDEX, LEN, and NINT are recognized as user-supplied function names.
- GENERIC statement: GENERIC means that generic names of functions supplied by VS FORTRAN will be recognized as generic; if GENERIC is not specified, the automatic function selection facility will not be in effect.
- IBM direct-access READ (*a' r [,b] [,ERR=c]*) [*list*] and WRITE (*a' r[,b]*) [*list*].
- PUNCH *b*, list.

Appendix B. Assembler Language Information

This appendix is intended to help you call the mathematical and service routines supplied with VS FORTRAN Version 2 from your assembler language program. It contains guidance information to help you call the mathematical and services routines of VS FORTRAN Version 2 and receive the results of their calculations.

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

You must bring the desired routines (modules) from the VS FORTRAN Version 2 library into main storage; usually you can do this with the load module. You can use 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) or IBM High Level Assembler/MVS & VM & VSE with the appropriate PTFs.

In Link Mode:

```
//jobname JOB ...operands
//ASM EXEC HLASMCLG,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 HLASMCLG,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.

Figure 64 on page 414 is an example of an Assembler language calling sequence with the CALL macro. Figure 65 on page 415 is an example of an Assembler language calling sequence with BALR.

In CMS and MVS, 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.

Note: Many of the mathematical routines require other mathematical routines and other service routines for some of their calculations.

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 *either* a list containing one 4-byte address constant for each argument, with the last address constant containing a 1 in its high order bit OR (where character arguments are used) a list containing one 4-byte address for each argument followed by a list containing one 4-byte address of a length 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.

Note: For calling assembler subprograms that access extended common data, different conventions are used. For more information on assembler subprograms and extended common, see the *VS FORTRAN Version 2 Programming Guide*.

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, bit, and character intrinsic routines, this information can be found in Figure 36 on page 281.

Argument information for nonmathematical routines can be found under Chapter 8, “Service Subroutines” on page 318.

Error messages resulting from incorrect arguments are explained in Appendix D, “Library Procedures and Messages” on page 438.

For information on passing correct parameters with extended addresses, see the *VS FORTRAN Version 2 Programming Guide*.

Examples of Assembler Language Calling Sequence

```
*      The following instructions show the use of a CALL
*      macro to call the library square root routine.
      .
      .
      CALL  S#QRT,(AMNT),VL          (See Notes 1, 2)
      STE  0,ANSWER
      .
      .
AMNT     DC    E'144'
ANSWER   DC    E'0'
      .
      .
```

Figure 64. Assembler Language Calling Sequence with CALL Macro

Notes to Figure 64:

- 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(S#QRT)
ANSWER2 DC     E'0'
      .
      .
ARG     DC     A(AMNX+X'80000000')
      .
      .
AMNX    DC     E'144'

```

Figure 65. Assembler Language Calling Sequence with BALR

Entry Names for VS FORTRAN Routines and Functions

The following lists contain routine entry names for VS FORTRAN Version 2 routines and functions.

Explicitly Called Routines

The following list contains routine entry names for explicitly called mathematical routines. The entry names marked with an asterisk (*) are also alternative mathematical library subroutines.

I#ABS, A#ABS, D#ABS, Q#ABS	I#MOD, A#MOD, D#MOD, Q#MOD
C#ABS*, CD#ABS*, CQ#ABS	A#NINT, D#NINT, N#INT, ID#NINT
C#OS*, D#COS*, Q#COS	T#AN* D#TAN*, Q#TAN
C#COS, CD#COS, CQ#COS	A#TAN*, DA#TAN*, QA#TAN
A#COS*, DA#COS*, QA#COS	A#TAN2*, DA#TN2*, Q#ATN2
C#OSH, D#COSH, Q#COSH	C#OTAN*, D#COTN*, Q#COTN,
S#IN*, D#SIN*, Q#SIN	T#ANH, D#TANH, Q#TANH
C#SIN, CD#SIN, CQ#SIN	E#XP*, D#EXP*, Q#EXP,
A#SIN*, DA#SIN*, QA#SIN	C#EXP, CD#EXP, CQ#EXP
S#INH, D#SINH, Q#SINH	S#QRT*, D#SQRT*, Q#SQRT
A#IMAG, D#IMAG, Q#IMAG	C#SQRT, CD#SQR, CQ#SQR
A#INT, D#INT, Q#INT	C#ONJG, D#CONJ, Q#CONJ
G#AMMA, D#GAMA,	I#DIM, A#DIM, D#DIM, Q#DIM
A#LGMA, D#LGMA	E#RF, D#ERF, Q#ERF
A#LOG*, D#LOG*, Q#LOG	E#RFC, D#ERFC, Q#ERFC
C#LOG, CD#LOG, CQ#LOG	D#PROD
A#LOG1*, D#LOG1*, Q#LOG1	I#SIGN, A#SIGN, D#SIGN, Q#SIGN

Implicitly Called Mathematical Routines

The following list contains routine entry names for implicitly called mathematical routines. More information on the functions provided by these names can be found in Figure 37 on page 310. The entry names marked with an asterisk (*) are also alternative mathematical library subroutines.

CDDVD#	FCXP8#
CDMPY#	FDXPD#*
CDVD#	FDXPI#
CMPY#	FDXP8#
CQDVD#	FIXPI#
CXMPR#	FIXP8#
CQMPY#	FQXPI#
FCDGD#	FQXPQ#
FCDXI#	FQXP2#
FCDX8#	FQXP8#
FCQCQ#	FRXPI#
FCQXI#	FRXPR#*
FCQX8#	FRXP8#
FCXPC#	F8XPI#
FCXPI#	F8XP8#

Explicitly Called Character Routines

The following list contains routine entry names for explicitly called character manipulation routines.

C#HAR	L#LGE
I#CHAR	L#LGT
L#EN	L#LLE
I#NDEX	L#LLT

Implicitly Called Character Routines

The following list contains routine entry names for implicitly called character manipulation routines. More information about these routines can be found in Figure 38 on page 313.

CCMPR#

CMOVE#

CNCAT#

Service Routines

Service subroutines can be called by the names and with the argument lists given in Chapter 8, “Service Subroutines” on page 318.

Explicitly Called Bit Functions

The following list contains routine entry names for explicitly called bit functions.

I#BCLR

I#NOT

I#BSET

I#AND

I#BTST

I#SHFT

I#OR

L#SHIFT

I#EOR

R#SHIFT

I#XOR

I#SHFTC

Mathematical Routine Results

Each mathematical routine returns a single answer of a type listed in Figure 36 on page 281.

- 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.

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 8, “Service Subroutines” on page 318.

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 69 on page 419 and Figure 70 on page 420. 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 66, 67, and 68 indicate which vector registers are used.

Figure 66. Vector Intrinsic Elementary Function Result Vector Registers

Function Type	Vector Registers Real Part	Vector Registers 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 67. Argument Vector Registers for Functions of One Argument

Argument Type	Vector Registers Real Part	Vector Registers Imaginary Part
REAL*4	14	
REAL*8	14-15	
COMPLEX*8	12	14
COMPLEX*16	12-13	14-15

Figure 68. Argument Vector Registers for Functions of Two Arguments

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	

Vector Count Register: Load the vector count register with 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 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.

To use vector intrinsic elementary functions with the mask mode on, use the entry name beginning with “W” in Figure 71 on page 420. The vector mask register will be preserved across the call.

To use vector intrinsic elementary functions with the mask mode off, use the entry name beginning with “V” in Figure 71 on page 420. In this case, the vector mask mode will be off on return, and the vector mask register is undefined on return.

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

If you enable fixed point overflow exception in the program mask, fixed point overflow can occur during the calculations in some of the routines. Initializing 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.

Examples of Assembler Language Calling Sequence

	.		
	.		
	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 69. Calling Sequences with Vector Mask Mode Off

Notes to Figure 69:

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), it will be identified and results returned to the Assembler caller in vector register 0.

```

      .
      .
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 70. Assembler Language Calling Sequences with Vector Mask Mode On

Notes to Figure 70:

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.

Entry Names for Vector Intrinsic Elementary Functions

For each of the mathematical functions listed in column 1 of Figure 71, there is a corresponding vector intrinsic elementary function that receives its arguments and returns its results in vector registers.

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.

Figure 71 (Page 1 of 3). Vector Entry Points for Mask Mode On and Mask Mode Off

Vector Entry If Mask Mode Off	Vector Entry If Mask Mode On
V#ACOS	W#ACOS
V#ASIN	W#ASIN
V#ATAN	W#ATAN
V#ATAN2	W#ATAN2
V#BTEST	W#BTEST
VC#ABS	WC#ABS
VC#COS	WC#COS
VCD#ABS	WCD#ABS

Figure 71 (Page 2 of 3). Vector Entry Points for Mask Mode On and Mask Mode Off

Vector Entry If Mask Mode Off	Vector Entry If Mask Mode On
VCD#COS	WCD#COS
VCDDVD#	WCDDVD#
VCD#EXP	WCD#EXP
VCD#LOG	WCD#LOG
VCD#SIN	WCD#SIN
VCD#SQR	WCD#SQR
VCDVD#	WCDVD#
VC#EXP	WC#EXP
VC#LOG	WC#LOG
V#COS	W#COS
V#COSH	W#COSH
V#COTAN	W#COTAN
VC#SIN	WC#SIN
VC#SQRT	WC#SQRT
VD#ACOS	WD#ACOS
VD#ASIN	WD#ASIN
VD#ATAN	WD#ATAN
VD#ATAN2	WD#ATAN2
VD#COS	WD#COS
VD#COSH	WD#COSH
VD#COTN	WD#COTN
VD#ERF	WD#ERF
VD#ERFC	WD#ERFC
VD#EXP	WD#EXP
VD#GAMMA	WD#GAMMA
VD#LGAMA	WD#LGAMA
VD#LOG	WD#LOG
VD#LOG10	WD#LOG10
VD#SIN	WD#SIN
VD#SINH	WD#SINH
VD#SQRT	WD#SQRT
VD#TAN	WD#TAN
VD#TANH	WD#TANH
V#ERF	W#ERF
V#ERFC	W#ERFC
V#EXP	W#EXP
VCD#CD#	WCD#CD#
VCD#XI#	WCD#XI#
VCX#PC#	WCX#PC#
VCX#PI#	WCX#PI#
VDX#PD#	WDX#PD#
VDX#PI#	WDX#PI#

Figure 71 (Page 3 of 3). Vector Entry Points for Mask Mode On and Mask Mode Off

Vector Entry If Mask Mode Off	Vector Entry If Mask Mode On
VIXPI#	WIXPI#
VRXPI#	WRXPI#
VRXPR#	WRXPR#
V#GAMMA	W#GAMMA
V#IBCLR	W#IBCLR
V#IBSET	W#IBSET
V#LGAMMA	W#LGAMMA
V#LOG	W#LOG
V#LOG10	W#LOG10
V#SIN	W#SIN
V#SINH	W#SINH
V#SQRT	W#SQRT
V#TAN	W#TAN
V#TANH	W#TANH

Appendix C. Sample Storage Printouts

This appendix is intended to help you use the symbolic dumps produced by VS FORTRAN Version 2. It contains descriptions of the output format of symbolic dumps, and output examples for variable items, for array items, and for nonrecoverable failure.

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:

```
xxxxxxxxxxxxxxxxxxx yyy zzzzzzzz
```

where:

xxxxxxxxxxxxxxxxxxx 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 values listed in Figure 72 on page 424.

Figure 72. Valid Values for Data Type Field in SDUMP Output

Code	Represents	Length
I8	Integer	(8 bytes)
I4	Integer	(4 bytes)
I2	Integer	(2 bytes)
I1	Integer or byte	(1 bytes)
UI1	Unsigned	(1 bytes)
L8	Logical	(8 bytes)
L4	Logical	(4 bytes)
L2	Logical	(2 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)

Variable Character

The variable value printing scheme for character data is as follows:

```
xxxxxxxxxxxxxxxxxxx CHR
```

where:

xxxxxxxxxxxxxxxxxxx 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:

```
xxxxxxx  aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa *bbbbbbbbbbbbbbbb*
```

where:

xxxxxxx 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 character 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 character area.

Array

The array value printing scheme is as follows:

ARRAY: xxxxxxxx TYPE:yyy

where:

xxxxxxx is the array name.

yyy is the data type.

Valid values for yyy are listed in Figure 72 on page 424.

Array Specification

DIMENSION x: (yyyyyyy:zzzzzzz)aaa

where:

x is the dimension (from 1 to 7).

yyyyyyy is the lower bound.

zzzzzzz 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:

xxxxxxx(dim1,dim2,dim3,dim4,dim5,dim6,dim7)

where:

xxxxxxx 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.

where:

xxxxxxx identifies the called module.

yyyyyyy identifies the caller routine.

OP/SYS is the operating system: CMS, or MVS

aaaaaa is the offset into the program unit. If blanks appear, then the offset is not available.

bbbbbbbbb 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:

xxxxxxx is the calling module name.

yyyyyyy 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:

xxxxxxx 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 nonrecoverable failure.

Example 1. Variable Items

This program assigns values to variable items.

```
C
C   SAMPLE PROGRAM TO DEMONSTRATE SDUMP OF SCALAR VARIABLES
C
C   SPECIFY THE VARIABLE TYPES
C
C   COMPLEX*8 C2,C1
C   COMPLEX*16 C3
C   COMPLEX*32 C4
C   CHARACTER CH1,CH2*8
C   REAL*8 A,B
C   REAL*16 YYYY,ZZZZ
C   INTEGER*2 JJJ,KKK,000,PPP
C   INTEGER*8 JJ8,KK8
C   INTEGER*1 JJ1,KK1
C   UNSIGNED*1 US1, US2
C   LOGICAL*1 P
C   LOGICAL*2 Q2
C   LOGICAL*4 Q
C   LOGICAL*8 P8
C
C   ASSIGN THE VALUES
C
C   P = .TRUE.
C   P8 = .FALSE.
C   Q = .FALSE.
C   Q2 = .TRUE.
C   CH2 = 'ABCDEFGH'
C   CH1 = '1'
C   R = 32.3733
C   A = 5.5
C   NUMLTS = 7
C   IABLSE = 6
C   ZZZZ = 4.0E5
C   IIII = 1111
C   JJJ = 222
C   JJ8 = 30286741648294625
C   KK8 = - 284347942294236
C   000 = 32767
C   PPP = -32768
C   JJ1 = 127
C   JJ1 = -120
C   US1 = 127
C   US2 = 250
C   B = 222.222
C   KKK = 999
C   LLL = 121212
C   MMM = 2147483647
C   NNN = -2147483647
C   YYYY = 25252525
C   C1 = (1.,1.)
C   C2 = (2.,2.)
C   C3 = (3.D0,3.D0)
C   C4 = (4.Q0,4.Q0)
20
C
C   PRINT MESSAGE AND INVOKE SDUMP
C
C   WRITE(6,*) " CALL SDUMP WITH SCALAR VARIABLES OF VARIOUS TYPES"
C   CALL SDUMP
10  STOP
END
```

Figure 73. Example 1—Source Program

Example 1. Output

The output is seen on the following page:

CALL SDUMP WITH SCALAR VARIABLES OF VARIOUS TYPES

```

SDUMP - SYMBOLIC DUMP FOR MODULE: MAIN#
MODULE MAIN# WAS CALLED BY OP/SYS .
MODULE MAIN# LAST CALLED AFBSDUMQ.
FROM OFFSET 000540 AT ISN. NO. 47.
NNN I4 -2147483647
-----
MMM I4 2147483647
-----
LLL I4 121212
-----
IIII I4 1111
-----
IABLSE I4 6
-----
NUMLTS I4 7
-----
R R4 0.323733E+02
-----
P8 L8 F
-----
Q L4 F
-----
Q2 L2 T
-----
P L1 T
-----
US2 UI1 250
-----
US1 UI1 127
-----
JJ1 I1 136
-----
KK8 I8
-----
JJ8 I8
-----
PPP I2 -32768
-----
000 I2 32767
-----
KKK I2 999
-----
JJJ I2 222
-----
ZZZZ R16 0.400000000000000000000000000000Q+06
-----
YYYY R16 0.252525250000000000000000000000Q+08
-----
B R8 0.22222200012207D+03
-----
A R8 0.55000000000000D+01
-----
CH2 1 CHR C1C2C3C4 C5C6C7C8 *ABCDEFGH *
-----
CH1 1 CHR F1 *1 *
-----
C4 C32 0.400000000000000000000000000000Q+01 0.400000000000000000000000000000Q+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 74. Example 1—Output

Example 2. Array Items

This program assigns values to array items.

```
C
C   SAMPLE PROGRAM TO DEMONSTRATE SDUMP OF ARRAY VARIABLES
C
C   SPECIFY THE VARIABLE TYPES
C
      COMPLEX*8 C2(5), C1(5)
      COMPLEX*16 C3(5)
      COMPLEX*32 C4(5)
      CHARACTER CH1(5), CH2(5)*8
      REAL*8 A(5), B(5)
      REAL*16 YYYY(5), ZZZZ(5)
      INTEGER*2 JJJ(5), KKK(5)
      INTEGER*8 JJ8(3)
      INTEGER*1 JJ1(2) /34, -23/, KK1(2) /Z7F, 0'12'/
      UNSIGNED INTEGER*1 US1(2,2)
      INTEGER NUMLTS(5), IABLSE(5), IIII(5), LLL(5)
      LOGICAL*1 P(5)
      LOGICAL*2 Q2(2) /T,F/
      LOGICAL*4 Q(4) /3*T, F/
      LOGICAL*8 P8(5)
C
C   ASSIGN VALUES
C
      P(1) = .TRUE.
      P(2) = .FALSE.
      P(3) = .TRUE.
      P(4) = .FALSE.
      P(5) = .TRUE.
      P8(1) = .TRUE.
      P8(2) = .FALSE.
      P8(3) = .FALSE.
      P8(4) = .FALSE.
      P8(5) = .TRUE.
      CH2(1)(:) = 'ABCDEFGH'
      CH2(2)(:) = 'ABCIJKLM'
      CH2(3)(:) = 'ABCNOPQR'
      CH2(4)(:) = 'ABCSTUVW'
      CH2(5)(:) = 'ABCXYZAB'
      CH1(1)(:) = '1'
      CH1(2)(:) = '2'
      CH1(3)(:) = '3'
      CH1(4)(:) = '4'
      CH1(5)(:) = '5'
      A(1) = 5.5
      A(2) = 4.5
      A(3) = 3.5
      A(4) = 2.5
      A(5) = 1.5
      NUMLTS(1) = 7
      NUMLTS(2) = 6
      NUMLTS(3) = 5
      NUMLTS(4) = 4
      NUMLTS(5) = 3
      IABLSE(1) = 6
      IABLSE(2) = 7
      IABLSE(3) = 8
      IABLSE(4) = 9
      IABLSE(5) = 10
```

Figure 75 (Part 1 of 2). Example 2—Source Program

```

JJ8(1) = 23439057897147378
JJ8(2) = "003405284FE8CA12"Z
JJ8(3) = - JJ8(2)
US1(1,1) = '10101010'B
US1(2,1) = '00011110'B
US1(1,2) = 45
US1(2,2) = 2
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 SDUMP
C
WRITE(6, *) " CALL SDUMP WITH ARRAY VARIABLES OF VARIOUS TYPE"
CALL SDUMP
STOP
END

```

Figure 75 (Part 2 of 2). Example 2—Source Program

Example 2. Output

The output is as follows:

CALL SDUMP WITH ARRAY VARIABLES OF VARIOUS TYPE

SDUMP - SYMBOLIC DUMP FOR MODULE: MAIN#
MODULE MAIN# WAS CALLED BY OP/SYS .
MODULE MAIN# LAST CALLED AFBSDUMQ.
FROM OFFSET DF8 AT ISN. NO. 114.

ARRAY: P8 TYPE:L8

DIMENSION 1: (1: 5)

P8(#)

# = 1	T
# = 2	F
# = 3	F
# = 4	F
# = 5	T

ARRAY: P TYPE:L1

DIMENSION 1: (1: 5)

P(#)

# = 1	T
# = 2	F
# = 3	T
# = 4	F
# = 5	T

ARRAY: LLL TYPE:I4

DIMENSION 1: (1: 5)

LLL(#)

# = 1	212
# = 2	312
# = 3	412
# = 4	512
# = 5	612

ARRAY: IIII TYPE:I4

DIMENSION 1: (1: 5)

IIII(#)

# = 1	1111
# = 2	3211
# = 3	4311
# = 4	6511
# = 5	1541

ARRAY: IABLSE TYPE:I4

DIMENSION 1: (1: 5)

IABLSE(#)

# = 1	6
# = 2	7
# = 3	8
# = 4	9
# = 5	10

ARRAY: NUMLTS TYPE:I4

DIMENSION 1: (1: 5)

NUMLTS(#)

# = 1	7
# = 2	6
# = 3	5
# = 4	4
# = 5	3

Figure 76 (Part 1 of 4). Example 2—Output

```

-----
ARRAY:  US1  TYPE:UI1
  DIMENSION 1: (      1:      2)
  DIMENSION 2: (      1:      2)
  US1( 1, #)
    # = 1      170
    # = 2      45
  US1( 2, #)
    # = 1      30
    # = 2       2
-----

ARRAY:  JJ8  TYPE:I8
  DIMENSION 1: (      1:      3)
  JJ8( #)
    # = 1      23439057897147378
    # = 2      14642369486440978
    # = 3     -14642369486440978
-----

ARRAY:  KKK  TYPE:I2
  DIMENSION 1: (      1:      5)
  KKK( #)
    # = 1      899
    # = 2      799
    # = 3      699
    # = 4      599
    # = 5      499
-----

ARRAY:  JJJ  TYPE:I2
  DIMENSION 1: (      1:      5)
  JJJ( #)
    # = 1      212
    # = 2      242
    # = 3      232
    # = 4      252
    # = 5      262
-----

ARRAY:  ZZZZ TYPE:R16
  DIMENSION 1: (      1:      5)
  ZZZZ( #)
    # = 1      0.40000000000000000000000000000000Q+06
    # = 2      0.40000000000000000000000000000000Q+04
    # = 3      0.40000000000000000000000000000000Q+03
    # = 4      0.40000000000000000000000000000000Q+02
    # = 5      0.40000000000000000000000000000000Q+01
-----

ARRAY:  YYYY TYPE:R16
  DIMENSION 1: (      1:      5)
  YYYY( #)
    # = 1      0.15151515000000000000000000000000Q+08
    # = 2      0.25252525000000000000000000000000Q+08
    # = 3      0.35353535000000000000000000000000Q+08
    # = 4      0.45454545000000000000000000000000Q+08
    # = 5      0.55555555000000000000000000000000Q+08
-----

ARRAY:  B  TYPE:R8
  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

```

Figure 76 (Part 2 of 4). Example 2—Output

```

-----
ARRAY:  A   TYPE:R8
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
-----

ARRAY:  CH2  TYPE:CHR
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      *
-----

ARRAY:  CH1  TYPE:CHR
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      *
-----

ARRAY:  C4   TYPE:C32
DIMENSION 1: (      1:      5)
C4( #)
# = 1      0.400000000000000000000000000000Q+01  0.400000000000000000000000000000Q+01
# = 2      0.300000000000000000000000000000Q+01  0.500000000000000000000000000000Q+01
# = 3      0.200000000000000000000000000000Q+01  0.600000000000000000000000000000Q+01
# = 4      0.100000000000000000000000000000Q+01  0.700000000000000000000000000000Q+01
# = 5      0.000000000000000000000000000000Q+00  0.800000000000000000000000000000Q+01
-----

ARRAY:  C3   TYPE:C16
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

```

Figure 76 (Part 3 of 4). Example 2—Output

```

ARRAY: C1  TYPE:C8
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

```

```

ARRAY: C2  TYPE:C8
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 76 (Part 4 of 4). Example 2—Output

Example 3. Nonrecoverable 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 does not 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 77. 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 438.
- 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 426.
- Control flow information appears last and is dependent on the SDUMP or TEST option. For a program unit active atabend 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 426.

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 sample output is obtained when the program is run without any run-time options specified.

Example 3. Output

This sample output is obtained when the program is run without 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 000020C).
DMSABN155T USER ABEND 0240 CALLED FROM 02F0D8.
```

Figure 78. Example 3—Output Under CMS or MVS Without Run-Time Options

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*, *IBM System/370 Extended Architecture Principles of Operation*, and *VM/ESA: ESA/Extended Configuration Principles of Operation*. For more on 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, floating-point, or decimal divide exceptions. These indicators can be interrogated dynamically by the subprograms described under Chapter 8, "Service Subroutines" on page 318.
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 Figure 36 on page 281, 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 439. There are seven program-interrupt messages: AFB112I, AFB116I, AFB117I, AFB207I, AFB208I, AFB209I, and AFB210I, listed in "Program-Interrupt Messages" on page 440. Execution errors are listed sequentially in "Execution Error Messages" on page 441.

All library messages are numbered. Operator messages are written when a STOP n or PAUSE statement is executed. Program-interrupt mes-

sages 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 xxxxxxxxxxxxxxxxx.

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 xxxxxxxxxxxxxxxxx.

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 vectored subprogram instead of floating-point data.

AFB117I VUNIN : PROGRAM INTERRUPT - VECTOR UNNORMALIZED OPERAND ON MULTIPLY, PSW xxxxxxxxxxxxxxxxx.

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 vectored subprogram instead of floating-point data.

AFB207I VFNTH|VINTH : PROGRAM INTERRUPT - [VECTOR] FLOATING-POINT OVERFLOW EXCEPTION, PSW xxxxxxxxxxxxxxxxx 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 $7.2 \times 10^{**}75$). For some negative exponents, implicitly called routines FRXPI#, FDXPI#, and FQXPI# issue this message.

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 UNDER- FLOW EXCEPTION, PSW xxxxxxxxxxxxxxxxx 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 $5.4 \times 10^{**}(-79)$).

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 (yyyyyy).

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 $n \neq 0$. 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.

Interruptions are explained in *IBM System/370 Extended Architecture Principles of Operation*.

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.

AFB083I PWATP: ORIGINATED TASK {Sxxxx | Uxxxx} WAS SCHEDULED BUT NOT WAITED FOR.

Explanation: An originated task was scheduled, but not waited for.

Standard Corrective Action: The scheduling routine waits for completion of the scheduled routine. When the scheduled routine completes, the scheduling routine resumes execution.

Programmer Response: Review program logic to ensure that a WAIT FOR statement is coded for all work that has completed.

AFB084I VASYP: ASYNCHRONOUS I/O ATTEMPTED IN ORIGINATED TASK ON UNIT nn.

AFB084I VIOUP: LONGLVL66 I/O ATTEMPTED IN ORIGINATED TASK ON UNIT nn.

AFB084I VOPEP: KEYED I/O ON TEMPORARY FILE ATTEMPTED IN ORIGINATED TASK ON UNIT nn.

Explanation: For format 1 of this message, an asynchronous I/O scheduling routine was called from within an originated task.

For format 2 of this message, a Fortran language level 66 I/O statement such as DEFINE FILE, PUNCH, FIND, or a READ or WRITE statement for level 66 direct access was encountered within an originated task.

For format 3 of this message, an OPEN statement for keyed access of a temporary file was attempted from within an originated task.

Supplemental Data Provided:

nn unit number

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file, and processing continues. Otherwise, the I/O statement is ignored, and processing continues.

Programmer Response: For format 1, correct the program so that no asynchronous routines are called from within an originated task.

For format 2, correct the program so that no Fortran level 66 I/O routines are called from within an originated task.

For format 3, correct the program so that no keyed I/O is performed on an unnamed temporary file from within an originated task.

AFB085I VEMGN | VEMG2: AN ERROR IS DETECTED WHILE EXECUTING UNDER ORIGINATED TASK {Sxxxx | Uxxxx}.

Explanation: An error was detected while executing under the control of an originated task for a parallel program.

Supplemental Data Provided

Sxxxx the sequence number of the originated task created by the ORIGINATE ANY TASK statement is xxxx.

Uxxxx the value of the originated task identifier provided on the ORIGINATE TASK statement is xxxx.

Standard Corrective Action: Execution continues and further messages are issued describing the error detected.

Programmer Response: Review the messages that follow and correct the problem causing the error. Error messages that follow may contain task identifiers using the same formatting described above.

AFB086I VOPEP: AN OPEN STATEMENT WITH A STATUS SPECIFIER OF ssssss WAS ISSUED FOR AN UNNAMED FILE ON UNIT nn.

Explanation: The STATUS specifier ssssss on the OPEN statement for unit nn was specified as OLD, NEW, or UNKNOWN by a subroutine executing under an originated task.

Standard Corrective Action: The unit is no longer connected to a file, and processing continues.

Programmer Response: If I/O to an unnamed file is intended, then either omit the STATUS specifier, or specify STATUS='SCRATCH'. Otherwise, provide the name of a file on the FILE specifier to perform I/O on a named file.

AFB087I VINTP: PARALLEL ENVIRONMENT INITIALIZED USING ONE VIRTUAL PROCESSOR.

Explanation: The PARALLEL run-time option was not specified, and a parallel language construct, or library lock or event service has been encountered.

Standard Corrective Action: Program execution continues. The parallel processing environment is initialized for execution with one virtual processor.

Programmer Response: Specify PARALLEL as a run-time option.

- AFB088I PLOKP: LOCK /// MAY NOT BE TERMINATED BECAUSE IT HAS NOT BEEN ORIGINATED, OR IS OWNED BY ANOTHER PARALLEL TASK.**
- AFB088I PEVTP: EVENT eeee MAY NOT BE TERMINATED BECAUSE IT HAS NOT BEEN ORIGINATED, OR IS OWNED BY ANOTHER PARALLEL TASK.**
- AFB088I PEVTP: EVENT eeee MAY NOT BE TERMINATED BECAUSE A CYCLE HAS NOT YET BEEN COMPLETED.**

Explanation: For format 1 of this message, the PLTERM service routine was passed lock identifier /// which is either owned by a parallel task other than the currently executing one, or has not been created by the PLORIG service routine, or has been previously terminated.

For format 2 of this message, the PETERM service routine was passed event identifier eeee which is either owned by a parallel task other than the currently executing one, or has not been created by the PLORIG service routine, or has been previously terminated.

For format 3 of this message, the event eeee is being terminated before its post count or wait count is satisfied.

Standard Corrective Action: The statement is ignored and execution continues.

Programmer Response: For format 1, ensure that the lock has been originated by the PLORIG service routine under the same parallel task that it is being terminated.

For format 2, ensure that the event has been originated by the PEORIG service routine under the same parallel task that it is being terminated.

For format 3, ensure that once the event cycle is started, it completes before it is terminated.

- AFB089I PLOKP: LOCK /// MAY NOT BE TERMINATED BECAUSE IT IS CURRENTLY LOCKED UNDER PARALLEL TASK {Sxxxx | Uxxxx }.**
- AFB089I PLOKP: LOCK /// MAY NOT BE RELEASED BECAUSE IT WAS NOT OBTAINED.**
- AFB089I PLOKP: LOCK /// WAS OBTAINED BUT NOT RELEASED.**

Explanation: For format 1 of this message, a CALL PLTERM statement was issued for lock /// which is currently locked under originated task xxxx or under the root task whose parallel task identifier is 0000.

For format 2 of this message, a CALL PLFREE statement was issued for lock ///. The lock is not currently locked.

For format 3 of this message, a lock was obtained using the PLLOCK service routine, and was not freed using

the PLFREE service routine before completion of a parallel loop, parallel section, parallel call, or scheduled subroutine.

Standard Corrective Action: The statement is ignored and execution continues.

Programmer Response: For format 1, ensure that the lock is freed before calling the PLTERM service routine.

For format 2, ensure that the lock was obtained by the PLLOCK service routine before calling the PLFREE service routine.

For format 3, the PLFREE service routine should be coded to free the lock prior to completion.

- AFB090I VOPEP: DIRECT I/O ATTEMPTED ON STRIPED FILE fffffff.**
- AFB090I VASYP: ASYNCHRONOUS I/O ATTEMPTED ON STRIPED FILE fffffff.**

Explanation: For the first format, an OPEN statement was coded for an unnamed file with ACCESS='DIRECT' UNIT=nn, and a file definition statement was provided for FTnnP001. For the second format, an asynchronous READ or WRITE statement was coded, and a file definition was provided for FTnnP001.

Supplemental Data Provided:

ffffff the striped file name

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing continues.

Programmer Response: If you intend to do direct or asynchronous I/O processing, either change the program to specify a different unit for which a striped file definition statement has not been provided, omit the file definition statement for FTnnP001, or provide a different ddname on the file definition statement for the non-striped file on which you want to do direct I/O.

- AFB091I VPIOS: BAD BLOCK READ FROM FILE fffffff.**

Explanation: An attempt was made to access a striped file, and the RECFM parameter on the file definition statement had a value of:

1. V/VS/D/VB/VBS/DB and the block descriptor does not match the actual block size.
2. V/VS/D/VB/VBS/DB and the record descriptor implies that the record extends past the end of the block.
3. VBS and a record descriptor value is too small; that is, it implies no data but is not a valid VBS null segment.
4. D and the actual block size does not exceed the specified block preface size.

5. FB with a block preface and the actual block size does not exceed the specified block preface size.
6. FB and the last record extends beyond the end of the block.

This may be because either the records are blocked and the final record in the block is incomplete, or because a block prefix was specified (BUFOFF) and the block size does not exceed the prefix size.

Supplemental Data Provided:

ffffff the striped file name

Standard Corrective Action: The I/O request is ignored and execution continues.

Programmer Response: Ensure that the file mode is read in the same way that it is written. Ensure that the length of the physical record is consistent with the default or specified block size.

AFB092I VPIOS: DATA STRIPES ARE INCONSISTENTLY DEFINED FOR STRIPED FILE fffffff.

Explanation: An attempt was made to access a striped file, and the file did not have all its stripes consistently defined in the file definition statements. The parameters that must match on the file definition statements for each parameter are:

- RECFM
- LRECL
- BLKSIZE
- BUFOFF

Other relevant characteristics of the stripes that must match are:

- Restrictions to input-only or output-only
- Whether or not the file may be deleted when closed
- File disposition (OLD/MOD/NEW)

Supplemental Data Provided:

ffffff the striped file name

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing continues.

Programmer Response: Ensure that all stripes are consistently defined.

AFB093I VFIST | CFIST | VPIOS: DDNAMES AND FILE NAMES FOR STRIPED FILE fffffff DO NOT MATCH.

Explanation: An attempt was made to access a striped file, and one of the following situations was detected:

- A file definition for ddname FTnnPmmm did not

refer to a data set name (MVS) or to a file identifier (CMS) that ends in the form xxxPyyy.

- A file definition for ddname FTnnPmmm did refer to a data set name (MVS) or to a file identifier (CMS) that ends in the form xxxPyyy, and:
 - the stripe numbers mmm and yyy did not match,
 - the file identifiers except for the stripe number yyy were not identical for all the stripes of a file, or
 - the stripe number yyy and the maximum stripe number xxx did not have the same number of digits.

Supplemental Data Provided:

ffffff the striped file name

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing continues.

Programmer Response: Check the file definition statements for FTnnP001 - FTnnPmmm. Ensure that for each individual stripe, the stripe number in the ddname matches the stripe number in the file identifier. Ensure that the file identifiers, except for the stripe number, are identical for all the stripes.

AFB094I CFIST | VFIST | VPIOS: DATA STRIPING ATTEMPTED ON UNSUPPORTED FILE OR DEVICE, FILE fffffff.

Explanation: An attempt was made to access a striped file, and a file definition statement for one or more of the stripes refers to a file or device that is not supported by data striping.

Supplemental Data Provided:

ffffff the striped file name

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing continues.

Programmer Response: Check the file definition statements and ensure that no stripes have been defined on devices that are not supported by data striping. Data striping may be done only on DASD or tape, and may not be performed on the following:

- A VSAM file
- A Partitioned Data Set (PDS)
- The error message unit (e.g., unit 6)
- An in-stream (DD * or DD DATA) data set (MVS only)
- A system output (SYSOUT) data set (MVS only)

AFB095I VPIOS: ATTEMPT TO USE THE BACKSPACE STATEMENT ON STRIPED FILE fffffff IS NOT SUPPORTED.

AFB095I VSIO: ATTEMPT TO USE THE BACKSPACE STATEMENT ON A PDS MEMBER IS NOT SUPPORTED, FILE fffffff.

Explanation: For format 1, the BACKSPACE statement was coded for unit **nn**, and a file definition statement was provided for FT**nn**P001.

For format 2, the BACKSPACE statement was coded for a unit connected to a PDS member that was either being processed for output or had BUFNO greater than 1 specified on the DD statement.

Supplemental Data Provided:

ffffff the file name

Standard Corrective Action: The I/O request is ignored, and processing continues.

Programmer Response: Either modify the program so it does not use BACKSPACE, or do not use striped files or PDS members.

AFB096I VFINF : FILEINF ARGUMENT LIST IS IN AN INCORRECT FORMAT.

AFB096I VFINF : FILEINF ARGUMENT NO. nn CONTAINS AN INVALID KEYWORD.

AFB096I VFINF : AN INCORRECT VALUE WAS GIVEN FOR THE FILEINF PARAMETER 'pppppp'.

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 **pppppp** contained a value not allowed for **pppppp**.

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.

pppppp 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 **pppppp**, provide a value in the form that is allowed for the parameter **pppppp**.

AFB097I VALOP: ALLOCATION FOR "xxxxxxx" FAILED. STORAGE NOT AVAILABLE

Explanation: Allocation for pointee array xxxxxxx was not successful.

Supplemental Data Provided:

xxxxxxx Pointee array name

Standard Corrective Action: All pointee arrays appearing before xxxxxxx in the ALLOCATE statement are allocated. xxxxxxx and those after it are not. Execution continues.

Programmer Response: Rerun the program with a larger region size.

AFB099I VOCMP : STATIC | DYNAMIC | EXTENDED COMMON aaaaaaa NOT AVAILABLE IN MAIN TASK PROGRAM FOR SHARING IN PARALLEL SUBROUTINE.

Explanation: A static, dynamic, or extended common block was not defined in the main task program before the service routine SHRCOM was called.

Supplemental Data Provided:

aaaaaaa static | dynamic | extended common block name

Standard Corrective Action: The request to share the static, dynamic, or extended common block is ignored and execution proceeds.

Programmer Response: Define the common block 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:

ffffff the name specified by the FILE specifier on the OPEN statement

Standard Corrective Action: The unit is no longer connected to a file, and processing 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 fffffff.**
- AFB103I VDYNA : ALLOCATION FAILED.
REQUEST FOR EXCLUSIVE USE OF A
SHARED DATA SET CANNOT BE
HONORED. FILE fffffff.**
- AFB103I VDYNA : ALLOCATION FAILED.
REQUESTED DATA SET NOT AVAIL-
ABLE. ALLOCATED TO ANOTHER JOB.
FILE fffffff.**
- AFB103I VDYNA : ALLOCATION FAILED. SPECI-
FIED VOLUME IS NOT MOUNTED. FILE
ffffff.**
- AFB103I VDYNA : ALLOCATION FAILED. INCOR-
RECT DEVICE NAME SUPPLIED. FILE
ffffff.**
- AFB103I VDYNA : ALLOCATION FAILED.
VOLUME DOES NOT HAVE ENOUGH
SPACE FOR THE DIRECTORY.
FILE fffffff.**
- AFB103I VDYNA : ALLOCATION FAILED. DIREC-
TORY SPACE REQUESTED IS LARGER
THAN THE PRIMARY.
FILE fffffff.**
- AFB103I VDYNA : ALLOCATION FAILED.
REQUIRED CATALOG IS NOT AVAIL-
ABLE. FILE fffffff.**
- AFB103I VDYNA : ALLOCATION FAILED. INSUF-
FICIENT SPACE IN CATALOG. FILE
ffffff.**
- AFB103I VDYNA : ssssssss FAILED.
ERROR CODE xxxx,
INFORMATION CODE yyyy.
FILE fffffff.**

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:

ffffff file name or data set name

ssssssss ALLOCATION or DEALLOCATION

xxxx error code

yyyy information code

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing 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*, or *OS/VS2 MVS System Programming Library: Job Management*. 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 unit is no longer connected to a file, 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 fffffff.

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 fffffff. 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 ffffffff NOT ALLOWED ON THE OPEN STATEMENT.

Explanation: A default ddname for an unnamed file cannot be specified on the FILE= specifier.

Supplemental Data Provided:

fffffff the default ddname or file name used (FTnnFmmm, FTnnPmmm, FTnnKkk, FTERRsss, FTPRTsss, FTUExxxx, FTUPxxxx, FTURxxxx, FTUQxxxx, FTSExxxx, FTSPxxxx, FTSRxxxx, FTSQxxxx).

Standard Corrective Action: The unit is no longer connected to a file, and processing continues.

Programmer Response: Either omit the FILE= specifier if you intend to do I/O to an unnamed file, or use a ddname that is not one of the default ddnames if you intend to do I/O to a named file.

-
- AFB108I name1 : STATUS OF 'NEW' IS NOT ALLOWED ON THE OPEN STATEMENT FOR AN EXISTING FILE, FILE ffffffff.**
- AFB108I name2 : STATUS OF 'NEW' IS NOT ALLOWED ON THE OPEN STATEMENT FOR A FILE ON A DEVICE THAT IS RESTRICTED TO INPUT ONLY, FILE ffffffff.**
- AFB108I name1 : STATUS OF 'OLD' IS NOT ALLOWED ON THE OPEN STATEMENT**

FOR A FILE WHICH DOES NOT EXIST, FILE ffffffff.

AFB108I name1 : ACTION OF 'READ' IS NOT ALLOWED ON THE OPEN STATEMENT FOR A FILE WHICH DOES NOT EXIST, FILE ffffffff.

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 ffffffff.

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 ffffffff.

AFB108I name2 : AN OUTPUT OPERATION IS BEING ATTEMPTED FOR A FILE ON A DEVICE THAT IS RESTRICTED TO INPUT ONLY, FILE ffffffff.

AFB108I name5 : ACTION OF 'READ' IS NOT ALLOWED ON THE OPEN STATEMENT WITH STATUS OF 'NEW', FILE ffffffff.

AFB108I name4 : AN OPEN STATEMENT FOR A FILE THAT DOES NOT EXIST IS NOT ALLOWED FOR A FILE ON A DEVICE THAT IS RESTRICTED TO INPUT ONLY, FILE ffffffff.

AFB108I VDIOS : STATUS OF 'OLD' OR 'UNKNOWN' IS NOT ALLOWED ON THE OPEN STATEMENT WITH ACCESS OF 'DIRECT' FOR A FILE WHICH IS EMPTY.

AFB108I name3: ACTION OF 'READWRITE' IS NOT ALLOWED ON THE OPEN STATEMENT FOR A FILE ON A DEVICE THAT IS RESTRICTED TO INPUT ONLY, FILE ffffffff

AFB108I VOPEP: POSITION OF 'pppppppp' IS NOT ALLOWED ON THE OPEN STATEMENT WITH ACCESS OF 'aaaaaaaa', FILE ffffffff

AFB108I VOPEP: DELIM OF 'dddddddd' IS NOT ALLOWED ON THE OPEN STATEMENT WITH ACCESS OF 'aaaaaaaa', FILE ffffffff

AFB108I VOPEP: DELIM OF 'dddddddd' IS NOT ALLOWED ON THE OPEN STATEMENT WITH FORM OF 'UNFORMATTED', FILE ffffffff

AFB108I VOPEP: PAD OF 'pppppppp' IS NOT ALLOWED ON THE OPEN STATEMENT WITH ACCESS OF 'aaaaaaaa', FILE ffffffff

AFB108I VOPEP: PAD OF 'pppppppp' IS NOT ALLOWED ON THE OPEN STATEMENT WITH FORM OF 'UNFORMATTED', FILE ffffffff

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. (On MVS an empty file will be treated as a new file. See the DASD file existence table for MVS in the VS FORTRAN Version 2 Programming Guide.)

For format 4 of this message, the OCSTATUS option is in effect and you attempted to connect a nonexistent file or an existing unformatted 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.

For format 9 of this message, the STATUS specifier on the OPEN statement was not specified, or was specified as STATUS='SCRATCH', or STATUS='UNKNOWN'. An attempt is being made to connect a file which does not exist on a device that is restricted to input only. Under CMS the attempt to connect the file is to a read-only disk. Under MVS the JCL specified LABEL=(,,IN).

For format 10 of this message, a file with no records in it was opened with STATUS other than "NEW" or "SCRATCH" for direct access. A file must contain formatted records in order to be used for direct access I/O.

For format 11 of this message, ACTION='READWRITE' 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 12 of this message, POSITION was specified on the OPEN statement with an ACCESS of 'DIRECT' or 'KEYED'. POSITION is allowed with sequential files only.

For format 13 of this message, DELIM was specified on the OPEN statement with an ACCESS of 'DIRECT' or 'KEYED'. DELIM is allowed with sequential files only.

For format 14 of this message, DELIM was specified on the OPEN statement with a FORM of 'UNFORMATTED'. DELIM is allowed with formatted files only.

For format 15 of this message, PAD was specified on the OPEN statement with an ACCESS of 'DIRECT' or 'KEYED'. PAD is allowed with sequential files only.

For format 16 of this message, PAD was specified on the OPEN statement with a FORM of 'UNFORMATTED'. PAD is allowed with formatted files only.

Supplemental Data Provided:

name1 VSIO, VDIOS, VKIOS, or VPIOS

name2 VSIO, VDIOS, or VPIOS

name3 VSIO or VDIOS

name4 VSIO or VPIOS

name5 VSIO, VDIOS, or VOPEP

ffffff file name

pppppppp POSITION value (APPEND, ASIS, or REWIND)

aaaaaaaa ACCESS value (DIRECT or KEYED)

dddddddd DELIM value (APOSTROPHE, QUOTE, or NONE)

pppppppp PAD value (YES or NO)

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing 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 mini-disk, 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.

For format 9 of this message, when running under CMS, find a READ/WRITE disk on which to connect the file if records are to be written, or ensure that the file exists if read operations are to be done. On MVS remove the LABEL=(,IN) statement from the JCL if records are to be written, or ensure the file exists if records are to be read.

For format 10 of this message, specify STATUS="NEW" or "SCRATCH" on the OPEN statement to cause records to be formatted on the file, preparing it for direct access I/O. Otherwise, make sure that the file is non-empty.

For format 11 of this message, either change the OPEN statement so that it does not specify ACTION='READWRITE', 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 12 of this message, make sure the ACCESS specifier accurately reflects the property of the file you are attempting to connect. If it does, remove the POSITION specifier from the OPEN statement.

For format 13 of this message, make sure the ACCESS specifier accurately reflects the property of the file you are attempting to connect. If it does, remove the DELIM specifier from the OPEN statement.

For format 14 of this message, make sure the FORM specifier accurately reflects the property of the file you are attempting to connect. If it does, remove the DELIM specifier from the OPEN statement.

For format 15 of this message, make sure the ACCESS specifier accurately reflects the property of the file you are attempting to connect. If it does, remove the PAD specifier from the OPEN statement.

For format 16 of this message, make sure the FORM specifier accurately reflects the property of the file you are attempting to connect. If it does, remove the PAD specifier from the OPEN statement.

AFB109I VINQP : A FILE NAME IS NOT ALLOWED ON THE INQUIRE STATEMENT WITH THE UNIT SPECIFIER, FILE=ffffff.

Explanation: An INQUIRE statement with both a FILE specifier with a non-blank value and a UNIT specifier is given.

Supplemental Data Provided: fffffff 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 VSIO\$: 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 fffffff.**
- AFB111I name : FILE DELETION FAILED. READ-ONLY DISK. FILE fffffff.**
- AFB111I name : FILE DELETION FAILED. UNEXPECTED ERASE RETURN CODE rc. FILE fffffff.**
- 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 VSIO\$, VDIOS, CVIOS, VVIO\$, or VKIOS
ffffff 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. ((*)RACF is a trademark of the International Business Machines Corporation.). Check the system completion code. For more information on system completion codes, see *MVS/370 Message Library: System Codes*, *MVS/XA Message Library: System Codes*, or *MVS/ESA System Codes*.

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 the ERASE command return codes can be found in *VM/SP CMS Command Reference*, *VM/XA SF CP Command Reference*, or *VM/SP System Messages and Codes*.

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* and *MVS/ESA VSAM Administration: Macro Instruction Reference*.

AFB112I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 440.

AFB113I VIADI : THE RECONNECT COMMAND REFERS TO A UNIT WHICH CANNOT BE RECONNECTED TO FILE fffffff.

Explanation: The IAD RECONNECT command was issued to reconnect a unit to a VSAM or non-sequential file.

Supplemental Data Provided:

ffffff 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:

fffffff file name

Standard Corrective Action: The unit is no longer connected to a file, and processing continues. If the ERR specifier was coded on the OPEN statement, processing 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:

fffffff file name

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement 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 440.

AFB117I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 440.

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 ffffffff 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: The unit is no longer connected to a file, and processing continues. 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
ffffff 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: The unit is no longer connected to a file, and processing continues. 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 VSIO | VDIO | VVIO | CVIO | VKIO :
sssssss STATEMENT IS NOT
ALLOWED WHEN THE FILE IS OPEN
WITH AN ACTION OF 'ddddddd'. FILE
ffffff.**

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:

sssssss the name of the incompatible statement.

ddddddd the value of the ACTION specifier that is in use.

ffffff 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 tttttt STATE-
MENT WHICH RESULTED IN
cccccccccccccccc CONDITION.
FILE fffffff.**

**AFB123I VSIO : READ STATEMENT IS NOT
ALLOWED FOLLOWING AN OUTPUT
STATEMENT ON A PDS MEMBER.
FILE fffffff.**

**AFB123I VSIO : WRITE STATEMENT IS NOT
ALLOWED FOLLOWING AN INPUT
STATEMENT ON A PDS MEMBER.
FILE fffffff.**

Explanation:

For format 1, 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.

For format 2, a READ statement followed an output operation without an intervening rewind.

For format 3, a WRITE statement followed an input operation without an intervening rewind.

Supplemental Data Provided:

sssssss

the name of the statement that was not allowed

tttttt

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

ffffff

the file name

Standard Corrective Action:

For format 1, execution continues, but the I/O request is ignored. If the ERR specifier was coded, control is passed to the indicated statement. Processing continues.

For format 2 and format 3, the I/O request is ignored, and processing continues.

Programmer Response:

For format 1, 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.

For format 2 and format 3, either code a REWIND statement before the READ or WRITE, or do not use PDS members.

AFB124I VKIOS : KEYID SPECIFIER ON THE READ STATEMENT HAS A VALUE OF <value> WHICH CONFLICTS WITH THE NUMBER OF KEYS IN THE KEYS SPECIFIER ON THE 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:

ffffff 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 the length in bytes of the search (or key) argument.

k 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 the length in bytes of the key being used.

ffffff 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:eeee.]
[xxxxxx SPECIFIER VALUE IS
vvvvvvvvvvv.]**

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:

ffffff the name of the file for which the READ statement was issued.

k 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:eeee** information are omitted if the KEYS specifier of the OPEN statement specified only one key or was not coded.)

sssss the starting position in each record of the key being used.

eeee the ending position.

xxxxxx KEY, KEYGE, or KEYGT—whichever specifier was used in the READ statement.

vvvvvvvvvv 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 the name of the input/output statement—for example, READ, REWRITE, DELETE.

nn 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 the name of the input/output statement.

ffffff 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 either WRITE or REWRITE.

nnnnn the length of the record that was built.

ffffff 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 the name of the statement being processed.

ffffff the name of the file involved in the input/output operation.

mmmmm the name of the VSAM macro that was issued (GET, PUT, POINT, and so on).

rc the VSAM return code.

hc the VSAM error feedback code in hexadecimal.

dc the same code in decimal.

fc the function code in hexadecimal.

You can find an explanation of the codes in *OS/VS Virtual Storage Access Method (VSAM) Programmer's Guide*.

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing continues.

Programmer Response: Take the action given in the appropriate manual.

AFB131I VKIOS : CONFLICTING DDNAMES WOULD BE REQUIRED FOR FILE fffffff 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 gen-

erate file names, it would duplicate the file name given in the OPEN statement.

Supplemental Data Provided:

ffffff the name of the file.

k the number of keys specified in the KEYS specifier of the OPEN statement.

Standard Corrective Action: The unit is no longer connected to a file, and processing continues. 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 fffffff 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 the file name.

f2 the related file.

r the record length of the file.

r2 the record length of the related file.

Standard Corrective Action: The unit is no longer connected to a file, and processing continues. 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 CMS) 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 fffffff 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: fffffff is the ddname of the empty file.

Standard Corrective Action: The unit is no longer connected to a file, and processing continues. 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:eeee, 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:

ffffff the name of the file, specified explicitly or taken by default, in the OPEN statement.

sssss the starting position in each record of the key to be used; **eeee** is the ending position.

Standard Corrective Action: The unit is no longer connected to a file, and processing continues. 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 CMS). (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:eeee, 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:

ffffff 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:eeee the position in the record of the key of reference.

vvvvvvvvvv 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 | POSITION | DELIM | PAD} SPECIFIER ON THE OPEN STATEMENT. UNIT nn.

Explanation: The ACTION specifier on the OPEN statement specified a value other than READ, WRITE, or READWRITE; or the POSITION specifier on the OPEN statement specified a value other than REWIND, APPEND, or ASIS; or the DELIM specifier on the OPEN statement specified a value other than APOSTROPHE, QUOTE, or NONE; or the PAD specifier on the OPEN statement specified a value other than YES or NO.

Supplemental Data Provided: **nn** is the unit number specified in the OPEN statement.

Standard Corrective Action: The unit is no longer connected to a file, and processing continues. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the specifier to a valid value.

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: The unit is no longer connected to a file, and processing continues. 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: The unit is no longer connected to a file, and processing continues. 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 (**ffffff**) 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 (ffffff) 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 VINTP : 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 or MVS/ESA system. Execution is impossible in this case, because several library routines must run at an address below 16 Mb.

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 VINTP : 'nnnnnnnn' IS AT LEVEL
'lbcml-lvl' BUT 'mmmmmmmm' IS AT
LEVEL 'mod-lvl'.**

Explanation: You were running your program in load mode, which requires loading the composite module(s). The module loaded (mmmmmmmm) was from a different release level of the Library than the first composite module loaded, (nnnnnnnn). If you are running

under CMS, the composite modules may be in a shared segment.

Supplemental Data Provided: The name of the loaded composite module (mmmmmmmm), and its level (mod-lvl). The name of the first composite module loaded (nnnnnnnn) and its level (lbcml-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 the 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 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/EXTENDED COMMON
aaaaaaaa. ABEND CODE ccc, REASON
CODE rr.**

**AFB143I VDIVP : DATA-IN-VIRTUAL SERVICE
dddddd FAILED FOR
DYNAMIC/EXTENDED COMMON
aaaaaaaa. COMPLETION CODE ccc,
REASON CODE rr.**

**AFB143I VDIVP : DATA-IN-VIRTUAL SERVICE
dddddd FAILED FOR LACK OF HARD-
WARE 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/extended common block name or (NONE), which indicates that no dynamic/extended 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*, or in *MVS/XA Message Library: System Codes*.

AFB144I VDIVP : DATA IN VIRTUAL SERVICES ARE NOT AVAILABLE ON THIS SYSTEM. VDIVP : DATA IN VIRTUAL SERVICES ARE NOT ALLOWED IN A PARALLEL SUBROUTINE. VDIVP : DATA IN VIRTUAL SERVICES ARE NOT AVAILABLE IN AN ORIGINATED TASK.

Explanation: For format 1, a DIV utility subroutine was invoked on a system that does not support data-in-virtual (that is, on CMS or on non-XA MVS or non-ESA MVS).

For format 2, a DIV utility subroutine was invoked by a parallel subroutine (using MTF).

For format 3, a DIV utility subroutine was invoked by an originated task in a parallel program.

Standard Corrective Action: Processing resumes and the service call is ignored.

Programmer Response: For format 1, remove the DIV utility subroutine calls or run the program on an MVS/XA or MVS/ESA system which has data-in-virtual support.

For format 2, remove the data-in-virtual utility subroutine calls from any parallel subroutines if using the multi-tasking facility (MTF). All of the DIV subroutine calls must be made from the main task program.

For format 3, remove the data-in-virtual utility subroutine calls from any subroutines that execute under an originated task. All of the DIV subroutine calls must be made from the root task.

AFB145I VINTP : 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 discontinuous 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 CMS and VS FORTRAN Version 2 Installation and Customization for MVS*.

If you are executing under CMS and the system name of a discontinuous 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 non-shareable-
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, yyyyyyyyyyyyyyy) 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 CMS) or the link pack area (in MVS) may have to be updated.

AFB150I name : OPEN NOT ALLOWED FOR A
MEMBER OF A PDS WITH 'DISP=MOD'
FILE fffffff

Explanation: A member of a PDS was to be opened for output and either the POSITION specified on the OPEN statement has a value of APPEND, or the allocation of the file specified DISP=MOD. PDS members cannot be extended.

Supplemental Data Provided: fffffff is the name of the file for which the OPEN statement was issued.

Standard Corrective Action: The unit is no longer connected to a file, and processing continues. If the ERR specifier was coded, control is passed to the indicated statement.

Programmer Response: Either code the program to rewrite the data as a new member or do not use PDS members.

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	the number of records formatted on the file
llll	the length of the records
ffffff	the name of the file

Standard Corrective Action: None.

Programmer Response: None.

AFB152I VSIO | 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: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing continues. 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 VINTP : 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 VDIPV : 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.

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) compile-time 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 unit is no longer connected to a file, and processing continues. 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 VOCMP : INSUFFICIENT VIRTUAL STORAGE FOR COMMON BLOCK 'common-name'.

Explanation: There is not enough virtual storage available for the common block 'common-name'.

Supplemental Data Provided: 'common-name' is the name of the common block.

Standard Corrective Action: The common block is not made available to one or more program units. The results of execution are unpredictable.

Programmer Response: Rerun the program in a larger region or virtual machine, or recompile all program units that refer to common block 'common-name' specifying a smaller length for this common block.

AFB157I VOCMP : SHRCOM WAS CALLED FROM A PARALLEL SUBROUTINE.
AFB157I VOCMP : SHRCOM WAS CALLED WITH AN INVALID COMMON BLOCK NAME.
AFB157I VOCMP : SHRCOM WAS CALLED WITH AN ARGUMENT LIST 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).

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 VOCMP : LENGTHS OF COMMON BLOCK aaaaaaaa ARE NOT CONSISTENT IN ALL PROGRAM UNITS.
AFB158I VOCMP : ATTRIBUTES OF COMMON BLOCK aaaaaaaa ARE NOT CONSISTENT IN ALL PROGRAM UNITS.

Explanation: For the first format of this message, different program units defined the common block aaaaaaaa with different lengths.

For the second format of this message, the attribute of a common block in a program unit is found to be different from an earlier reference to the same common block name by a different program unit.

Supplemental Data Provided: aaaaaaaa is the name of the common block.

Standard Corrective Action: The common block is not made available to one or more program units. The results of execution are unpredictable.

Programmer Response: For the first format of the message, assure that the length of common block aaaaaaaa is the same in all program units that refer to it.

For the second format of the message, compile all program units that refer to the common block aaaaaaaa with the same common option specifying this common block name, or compile all program units without any of them.

AFB159I name : ARGUMENT {2 | 3} TO function-name IS INVALID.

Explanation: The indicated 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; for ISHFTC, the result = 0; for RSHIFT, the result = 0; for LSHIFT, the result = 0; for IBITS, the result = 0.

Programmer Response: Specify the invalid 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.

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.

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 (ffffff) 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.

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 (ffffff) 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:

ffffff name of the file for which the OPEN statement was issued.

Standard Corrective Action: The unit is no longer connected to a file, and processing continues.

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 IGNORED FOR A VSAM FILE. FILE fffffff.

AFB166I VVIOS: ENDFILE STATEMENT IS IGNORED FOR A PDS MEMBER. FILE fffffff.

Explanation: The ENDFILE statement was coded for a unit connected to either a VSAM file or a PDS member.

Supplemental Data Provided:

ffffff the file name

Standard Corrective Action: The I/O request is ignored, and processing continues.

Programmer Response: Either modify the program so it does not use ENDFILE, or do not use VSAM and PDS members.

**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 **xxxxxxx**, 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 (**xxxxxxx**) 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. FILE fffffff.
DISK IS NOT ACCESSED, DISK fm.**

**AFB169I CDYNA : FILEDEF FAILED. FILE fffffff.
ALL DDNAME COMBINATIONS HAVE
BEEN EXHAUSTED.**

**AFB169I CDYNA : FILEDEF FAILED. FILE fffffff.
UNEXPECTED ERROR CODE FROM
FSSTATE, ERROR CODE err.**

**AFB169I CDYNA : FILEDEF FAILED. FILE fffffff.
UNEXPECTED RETURN CODE FROM
FILEDEF, RETURN CODE rc.**

**AFB169I CDYNA : FILEDEF CLEAR FAILED. FILE
ffffff.**

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:

ffffff file name or data set name

aaaaaaaa FILEDEF or FILEDEF CLEAR

d CMS disk specified in the FILE specifier or the default disk

fm file mode (CMS)

nnn system return code for FILEDEF or error code for FSSTATE

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing 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* 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* 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.

AFB1711 VDIOS | VSIO : 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 (ffffff) 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.

AFB1721 VDIOS | VSIO : 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 (ffffff) specified in the OPEN statement.

Standard Corrective Action: The unit is no longer connected to a file, and processing continues.

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.

AFB1731 VDIOS | VSIO : 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 (ffffff) 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.

AFB1741 VDIOS | VSIO : 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 (ffffff) 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.

AFB1751 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 VSIO, VDIOS, VKIOS, or VOPEP
nnnn unit identifier

Standard Corrective Action: The statement is ignored, and processing continues.

Programmer Response: Correct the invalid unit number.

AFB1761 MVBITS: SUM OF ARGUMENT NUMBERS {2 AND 3 | 3 AND 5} TO "MVBITS" IS INVALID.

Explanation: The value of adding the second and third, or the third and fifth arguments to MVBITS is greater than 32, the maximum number of bits in the first and fourth arguments.

Supplemental Data Provided: The argument numbers in error are given and a traceback provided.

Standard Corrective Action: No bits are moved; execution continues.

Programmer Response: Supply arguments with the valid range.

AFB177I I8SIM: INVALID INTEGER*8 SIMULATION REQUEST. EXECUTION CANNOT CONTINUE.

Explanation: The extended precision integer (INTEGER*8) simulator has detected an invalid request.

Standard Corrective Action: Execution is terminated.

Programmer Response: Report the problem to IBM.

AFB178I I8SIM: PROGRAM CHECK OCCURRED DURING SIMULATION OF INTEGER*8 OR LOGICAL*8 INSTRUCTION.

Explanation: The extended precision integer (INTEGER*8) simulator has encountered an error during simulation of the INTEGER*8 computation. This could be caused by an operand out of bounds, for example.

Standard Corrective Action: Execution has terminated. A traceback will be provided.

Programmer Response: Examine the operands used in the INTEGER*8 calculation.

AFB179I VARGP: INCORRECT NUMBER/TYPE OF ARGUMENTS PASSED TO "ARGSTR" ROUTINE.

Explanation: The ARGSTR routine expects to receive 2 arguments. Only 1 was passed.

Standard Corrective Action: The argument provided is left unchanged and execution continues. A traceback is provided.

Programmer Response: Correct the call to ARGSTR by providing two valid arguments.

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, specified a default file name, or specified an invalid CMS file identifier or invalid MVS data set name for a dynamically allocated file.

Supplemental Data Provided:

nn	unit number for which the OPEN statement was issued.
xx	OPEN or INQUIRE
ffffff	file name

Standard Corrective Action: The unit is no longer connected to a file, and processing continues.

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: The unit is no longer connected to a file, 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 unit is no longer connected to a file, and processing continues.

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 unit is no longer connected to a file, and processing continues.

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 unit is no longer connected to a file, and processing continues.

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 (ffffff) for which the request was made.

Standard Corrective Action: The unit is no longer connected to a file, 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 SUB-ROUTINE (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.

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 III, 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 (III) 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 11, "Extended Error-Handling Topics" on page 382.

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 0 through 499 and 900 through 999 (VS FORTRAN Version 2 Library) and 500 through 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 500 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: III.

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 (III).

Standard Corrective Action: The function is not performed, and execution continues. Be aware that the result may differ between serial code and code located in an originated task.

Programmer Response: Specify the correct length.

AFB192I VASYP: ASYNCHRONOUS I/O DDNAME fffffff 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 (ffffff) 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: III.

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 (III).

Standard Corrective Action: The function is not performed, and execution continues.

Programmer Response: Specify the correct length.

AFB194I VASYP : ASYNCHRONOUS I/O DDNAME fffffff, 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 (ffffff) of the file on which asynchronous I/O was to be attempted.

xxx may be 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 | CNCAT : 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.

Standard Corrective Action: The assignment is not completed, and execution continues. Be aware that the result may differ between serial code and code located in an originated task.

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.

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.

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 11, "Extended Error-Handling Topics" on page 382.

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.

Standard Corrective Action: The concatenation operation is not performed.

Programmer Response: Specify the correct length.

AFB200I VVIOS : END OF INTERNAL FILE, PROCESSING ENDS.

Explanation: The end of an internal file was reached before the completion of an internal I/O request.

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 IRECL. THE FILE IS NOT COMPATIBLE WITH NON-FORTRAN ACCESS METHODS. FILE ffffffff.

AFB201I VIOUP : UNFORMATTED DIRECT ACCESS DATA IS LONGER THAN THE RECORD LENGTH OF IRECL. THE REMAINING DATA IS TRANSFERRED FROM | TO THE NEXT RECORD. FILE ffffffff.

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 than 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 ffffffff.

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 FORTRAN 77 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 CMS): 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.

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:

ffffff 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 (ffffff).

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.

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 and the smallest is -2**15 for INT*2 and -2**31 for INT*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 440.

AFB208I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 440.

AFB209I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 440.

AFB210I

Explanation: For information on this message, refer to "Program-Interrupt Messages" on page 440.

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 (ffffff).

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 or end of record, which means that you are out of record space.

Supplemental Data Provided: The name of the file (ffffff).

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 if there are more characters remaining on the I/O list.

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 (ffffff) 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 (ffffff).

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 (ffffff) 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 (ffffff) 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 (ffffff).

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 (ffffff).

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.

For a NAMELIST READ operation, an end-of-file condition may have resulted because of a missing end quote on a character data value or because the NAMELIST name in the input did not match the NAMELIST name in the READ statement.

Supplemental Data Provided:

name CVIOS, VSIVOS, VASYP, or VVIVOS.

ffffff the file name

Standard Corrective Action: The next file is read, that is, the data set sequence number is increased by 1 if the file being read is not a named file or within an originated task.

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.

For NAMELIST READ operations, ensure that all character data values are specified within quotes and that the NAMELIST name matches the name in the NAMELIST READ statement.

AFB218I name : I/O ERROR, FILE fffffff, gggggggg.**AFB218I name : I/O ERROR, FILE fffffff, ERROR OCCURRED WHILE PROCESSING THE BACKSPACE STATEMENT.****AFB218I name : I/O ERROR, FILE fffffff, ATTEMPTING TO READ FROM SYSOUT DATA SET.****AFB218I VPIOS: I/O ERROR, FILE fffffff, SYSTEM COMPLETION CODE ccc-rr.****AFB218I VPIOS: I/O ERROR, FILE fffffff, BSP MACRO RETURN CODE - REASON CODE ccc-rr.****AFB218I VPIOS: I/O ERROR, FILE fffffff, UNEXPECTED END OF FILE OCCURRED.****AFB218I VPIOS: I/O ERROR, FILE fffffff, UNEXPECTED CONCATENATION OCCURRED.****AFB218I VSIVOS: I/O ERROR, FILE fffffff, READING/WRITING AFTER END-OF-FILE ON A NAMED FILE.****AFB218I VSIVOS: I/O ERROR, FILE fffffff, READING/WRITING AFTER END-OF-FILE IN AN ORIGINATED TASK.**

Explanation: A storage media error may have occurred on either tape or disk after a previous access was successful.

For format 5, the error occurred when a BSP macro was executed.

For format 6, while positioning the file, an unexpected end of file was reported by the CHECK macro.

For format 7, while positioning the file, an unexpected file concatenation was reported by the CHECK macro.

For format 8, a READ or WRITE has been attempted after an end-of-file was encountered on a named file.

For format 9, a READ or WRITE has been attempted after an end-of-file was encountered on file in an originated task.

Supplemental Data Provided:

name VSIVOS, VASYP, VDIVOS or XSIVOS

ffffff file name

gggggggg

With MVS see MVS/DFP Macro Instructions for Data Sets for a description of the message buffer format in the discussion of SYNADAF. type of I/O error

ccc system completion code or BSP macro return code

rr system return code or BSP macro reason code

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing continues. 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 format 4, use the system completion code to determine the nature of the error.

For format 5, check the BSP error return code. For more information on these error codes, see *MVS/XA Data Administration: Macro Instruction Reference*,

MVS/370 Data Administration: Macro Instruction Reference, or MVS/ESA System—Data Administration.

For formats 6 and 7, refer the problem to the people at your installation who give system support for VS FORTRAN Version 2.

For READ operation, 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.

For WRITE operation, 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.

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 name : ssssssss FAILED. UNEXPECTED TAPECTL ERROR CODE er. FILE fffffff.

AFB219I CFIST : 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 : RECONNECT FAILED. MISSING OR INVALID CONTROL STATEMENT. FILE fffffff.

AFB219I VOPEP | VINQP : OPEN | INQUIRE 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, an error occurred when a TAPECTL macro instruction was executed. This only happens to the standard-labeled (SL) tape files with the LEAVE option specified in the FILEDEF command.

For format 5 (CMS only), an error (other than disk not accessed) occurred when an FSSTATE macro instruction was executed.

For format 6 (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 7, 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 8, 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 9, 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	VSIO, VDIO, 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 <i>OS/VS Message Library: VS2 System Codes</i> , or <i>MVS/XA Message Library: System Codes</i> .)
rr	Reason code
fr	FSSTATE error code
er	TAPECTL error code
ffffff	File name

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing 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 ISCI/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 TAPECTL error return code. For more information on these errors, see *VM/SP Command and Macro Reference*.

For format 5 of this message, check the FSSTATE error return code. For more information on these error codes, see *VM/SP Command and Macro Reference*.

For format 6 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 7 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 8 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 9 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 VSIO, VDIOS, DIOCS, CVIOS, VVIO, 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 SUB-
SCRIPT 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 |
BINARY | OCTAL} CHARACTER char.**

Explanation: An invalid character is encountered on input for the Z, B, or O format code, respectively.

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 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 (ffffff).

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 (ffffff).

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 VPARAM : 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.

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 (ffffff).

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 (ffffff).

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing continues.

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 VDIOS : 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 the unit number specified in the I/O statement.

aaaaaaaaaa either SEQUENTIAL or KEYED.

Standard Corrective Action: The I/O request is ignored.

Programmer Response:

- If you want to do direct I/O processing, the statement must 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 (ffffff).

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 (ffffff).

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 (ffffff).

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 (ffffff) 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.
ABEND REASON CODE= kkkkkkkk
SCB/SDWA=aaaaaaaa
IO cccccccc.
PSW=xxxxxxxxxxxxxxxxxx
ENTRY POINT=eeeeeee.
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
ARS 0 - 3 nnnnnnnn nnnnnnnn
nnnnnnnn nnnnnnnn
ARS 4 - 7 nnnnnnnn nnnnnnnn
nnnnnnnn nnnnnnnn
ARS 8 - 11 nnnnnnnn nnnnnnnn
nnnnnnnn nnnnnnnn
ARS 12 -15 nnnnnnnn nnnnnnnn
nnnnnnnn nnnnnnnn
FRGS 0 & 2 nnnnnnnn nnnnnnnn
nnnnnnnn nnnnnnnn
FRGS 4 & 6 nnnnnnnn nnnnnnnn
nnnnnnnn nnnnnnnn
DATA AT ADDRESS (xxxxxxx)
yyyyyyyy yyyyyyyy yyyyyyyy.
DYNAMIC COMMON MAP
dddddd AT fffffff (ggggggggg)
dddddd AT fffffff(gggggggg)
MAP FOR SHAREABLE LOAD MODULE:
bbbbbbbb
hhhhhhhh AT fffffff
hhhhhhhh AT fffffff
LOADED LIBRARY MODULES
jjjjjjj AT fffffff
jjjjjjj AT fffffff

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.

kkkkkkkk is the reason code (in hexadecimal) if a user abend occurred. This line will be printed only if the reason code is present.

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.

If dynamic common blocks have been used, a map of obtained COMMON areas is provided where **dddddd** is the name of the COMMON, **ffffff** 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 **ffffff** 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 **ffffff** 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*4 BASE=0, INTEGER*4
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*4 variables or 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*4
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 constant and J represents an INTEGER*4 variable or 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*4
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 constant and J represents an INTEGER*4 variable or 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 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.

ation 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*4 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*4 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 FCD CD : 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*4 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 AFBFCD CD (FCD CD#), the real part of the COMPLEX*16 exponent P, or in the subprogram AFBFCDXI (FCDXI#), the INTEGER*4 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 appropriate compensation. Bypass the exponentiation operation if necessary.

ation and make the appropriate compensation. Bypass the exponentiation operation if necessary.

AFB248I FQXPI : REAL*16 BASE=0.0, INTEGER*4 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*4 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 $Y \cdot \log_2(X)$ 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 $X=0$, $RESULT=-\bullet$;
 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
 $\pi \cdot (2^{**18})$**

Explanation: In the subprogram AFBSSCN (SIN and COS), the absolute value of an argument is greater than or equal to $2^{**18} \cdot \pi$ ($2^{**18} \cdot \pi = .823\ 550\ E+06$).

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 57.298degrees) 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.

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: $\sinh(X)=\pm\bullet$; $\cosh(X)=\bullet$

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 $x > 1.0$, $\text{ACOS}(x) = 0$;
 If $x < -1.0$, $\text{ACOS}(x) = \pi$;
 If $x > 1.0$, $\text{ASIN}(x) = \pi/2$;
 If $x < -1.0$, $\text{ASIN}(x) = -\pi/2$.

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 $\pi \cdot (2^{**18})$.**

Explanation: In the subprogram AFBSTNCT (TAN or COTAN), the absolute value of the argument is greater than or equal to $2^{**18} \cdot \pi$ ($2^{**18} \cdot \pi = .823\ 550\ E+6$).

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 57.2958degrees) 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 ($\pm\pi/2$, $\pm3\pi/2$, ... for the tangent or 0, $\pm\pi$, $\pm2\pi$, ... for the cotangent).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: $Result=\bullet$.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.2958degrees) 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 AFBLLLOG (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 logsub10|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.

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 1degrees) 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.

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: $DSINH(X)=\pm\bullet$;
 $DCOSH(X)=\bullet$.

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 $\pi \cdot (2^{}50)$.**

Explanation: In the subprogram AFBLTNCT (DTAN or DCOTAN), the absolute value of the argument is greater than or equal to $2^{**}50 \cdot \pi$ ($2^{**}50 \cdot \pi = .353\ 711\ 887\ 601\ 422\ 01D+16$).

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 1degrees) 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 ($\pm\pi/2$, $\pm3\pi/2$, ... for the tangent; 0, $\pm\pi$, $\pm2\pi$, ... for the cotangent).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result= \bullet .

Programmer Response: Make sure that the argument (in radians where 1 radian is equivalent to 57.295 779 513 1degrees) 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*32 BASE = (0.0,0.0), INTEGER*4 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*4 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= $\bullet+0i$;
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: $\text{Result} = \bullet(\cos X + i\sin X)$, 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 $\pi(2^{}18)$.**

Explanation: In the subprogram AFBCSEXP (CEXP), the absolute value of the imaginary part of the argument is greater than or equal to $2^{**}18\pi$ ($2^{**}18\pi = .823550\text{E}+6$).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If x is the real part of the argument, then $\text{result} = e^{**}x + 0i$, 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.

Standard Corrective Action: $\text{Result} = -\bullet + 0i$.

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 $\pi(2^{}18)$.**

Explanation: In the subprogram AFBCSSCN (CSIN or CCOS), the absolute value of the real part of the argument is greater than or equal to $2^{**}18\pi$ ($2^{**}18\pi = .823550\text{E}+6$).

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

$\text{CCOS Result} = \cosh(y) + 0i$;

$\text{CSIN Result} = 0 + \sinh(y)i$

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 57.2958degrees) 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, $\text{result} = \bullet/2 * (\sin X + i\cos X)$.

For cosine, $\text{result} = \bullet/2 * (\cos X - i\sin X)$.

If imaginary part < 0 (X is real portion of argument):

For sine, $\text{result} = \bullet/2 * (\sin X - i\cos X)$.

For cosine, $\text{result} = \bullet/2 * (\cos X + i\sin X)$.

Programmer Response: Make sure that the imaginary part of the argument (in radians, where 1 radian is equivalent to 57.2958degrees) 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: $\text{Result} = (\cos X + i \sin X)$, 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 2100.**

Explanation: In the subprogram AFBCQEXP (CQEXP), the absolute value of the imaginary part of the argument is greater than 2^{100} .

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If x is the real part of the argument, then $\text{result} = e^{x} + 0i$, 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.

Standard Corrective Action: $\text{Result} = -\bullet + 0i$.

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 2100.**

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, $\text{result} = 0 + \text{DSINH}(Y) * i$ and, for CQCOS, $\text{result} = \text{DCOSH}(Y) + 0 * i$.

Programmer Response: Make sure that the real part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1 degrees.) 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, $\text{result} = \bullet / 2 * (\sin X + i \cos X)$.

For cosine, $\text{result} = \bullet / 2 * (\cos X - i \sin X)$.

If imaginary part < 0 , (X is real portion of argument):

For sine, $\text{result} = \bullet / 2 * (\sin X - i \cos X)$.

For cosine, $\text{result} = \bullet / 2 * (\cos X + i \sin X)$.

Programmer Response: Make sure that the imaginary part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1 degrees) 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: $\text{Result} = \bullet(\cos X + i\sin X)$, 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 $\text{Result} = e+0i$, 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.

Standard Corrective Action: $\text{Result} = -\bullet+0i$.

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 $\pi(2^{}50)$.**

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 = $0 + D\sinh(Y) + i$; for CDCOS, the result = $D\cosh(Y) + 0i$.

Programmer Response: Make sure that the real part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1degrees) 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, $\text{result} = \bullet/2 * (\sin X + i\cos X)$.

For cosine, $\text{result} = \bullet/2 * (\cos X - i\sin X)$.

If imaginary part <0 , (X is real portion of argument):

For sine, $\text{result} = \bullet/2 * (\sin X - i\cos X)$.

For cosine, $\text{result} = \bullet/2 * (\cos X + i\sin X)$.

Programmer Response: Make sure that the imaginary part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1degrees) 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 (ffffff).

Standard Corrective Action: If an OPEN statement caused the error, the unit is no longer connected to a file and processing continues. Otherwise, the I/O statement is ignored and processing 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 (ffffff).

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 (ffffff).

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: $\text{Result} = |x|^{1/2}$.

Programmer Response: Make sure that the argument is within the allowable range. Either modify the argu-

ment, 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: $\text{Result} = \bullet$.

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 ($0 < X < 4.2937 \times 10^{73}$).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: $\text{Result} = \bullet$.

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: $\text{Result} = \bullet$.

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 $\log_{sub10}|X|$.

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 2100.**

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= $\text{SQRT}(2)/2$.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.295 779 513 1 degrees.) 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.

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: $\text{QSINH}(X)=\pm•$;
 $\text{QCOSH}(X)=•$.

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) = π ;
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 degrees) 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 ($\pm\pi/2$, $\pm 3\pi/2$, for the tangent; 0, $\pm\pi$, $\pm 2\pi$, 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 degrees.) 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 ($0 < x < 4.2937 \times 10^{**73}$).

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 sit-

uation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

AFB302I F8XPI : INTEGER*8 BASE=0, INTEGER*4 EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation ($I^{**}J$) in the subprogram AFBF8XPI (F8XPI#), where I represents an INTEGER*8 variable or constant and J represents an INTEGER*4 variable or constant, 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.

AFB303I FIXP8 : INTEGER*4 BASE=0, INTEGER*8 EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation ($I^{**}J$) in the subprogram AFBFIXP8 (FIXP8#), where I represents an INTEGER*4 variable or constant and J represents an INTEGER*8 variable or 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 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.

AFB304I F8XP8 : INTEGER*8 BASE=0, INTEGER*8 EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation ($I^{**}J$) in the subprogram AFBF8XP8 (F8XP8#), where I and J represent INTEGER*8 variables or 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.

AFB305I FRXP8 : REAL*4 BASE=0.0, INTEGER*8 EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation ($R^{**}J$) in the subprogram AFBFRXP8 (FRXP8#), where R represents a REAL*4 variable or constant and J represents an INTEGER*8 variable or 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.

AFB306I FDXP8 : REAL*8 BASE=0.0, INTEGER*8 EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO.

Explanation: For an exponentiation operation ($D^{**}J$) in the subprogram AFBFDXP8 (FDXP8#), where D represents a REAL*8 variable or constant and J represents an INTEGER*8 variable or 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 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.

AFB307I FQXP8 : REAL*16 BASE=0.0, INTEGER*8 EXPONENT=exponent, LESS THAN OR EQUAL TO 0.

Explanation: For an exponentiation operation ($Q^{**}J$) in the subprogram AFBFQXP8 (FQXP8#), where Q represents a REAL*16 variable or constant and J represents an INTEGER*8 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.

AFB308I FCXP8 : COMPLEX*8 BASE = (0.0,0.0), INTEGER*8 EXPONENT=exponent, LESS THAN OR EQUAL TO 0.

Explanation: For an exponentiation operation ($Z^{**}P$) where the complex*8 base Z equals 0, in the subprogram AFBFCXP8 (FCXP8#) the INTEGER*8 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.

**AFB309I FCDX8 : COMPLEX*16 BASE = (0.0,0.0),
INTEGER*8 EXPONENT=exponent, LESS
THAN OR EQUAL TO 0.**

Explanation: For an exponentiation operation ($Z^{**}P$) where the COMPLEX*16 base Z equals 0, in the sub-program AFBFCDX8 (FCDX8#), the INTEGER*8 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.

**AFB310I FCQX8 : COMPLEX*32 BASE = (0.0,0.0),
INTEGER*8 EXPONENT = exponent,
LESS THAN OR EQUAL TO 0.**

Explanation: For an exponentiation operation ($Z^{**}P$) where the COMPLEX*32 base Z equals 0, in the sub-program AFBFCQX8 (FCQX8#) the INTEGER*8 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.

**AFB311I VCVTH : EXTENDED INTEGER VALUE
OUT OF RANGE (nnnnnnnn).**

Explanation: An input integer was too large to fit into the integer data item. (The largest extended integer that can be processed is $2^{**}63-1$ and the smallest is $-2^{**}63$.)

Supplemental Data Provided: The input integer (nnnnnnnn).

Standard Corrective Action: The maximum positive or negative value will be returned.

Programmer Response: Make sure that all integer input data used is within the required range for the integer variable size.

**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 11, "Extended Error-Handling Topics" on page 382.

**AFB902I VEMGN : ERROR NUMBER nnnn
REQUESTED BY MODULE module-name
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 (084 through 951), 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 REFER-
ENCE 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 11, "Extended Error-Handling Topics" on page 382.

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 (ffffff).

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 VINTP : 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 initialize 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. II, REQUESTED BY MODULE mod-
name HAS NO MESSAGE SKELETON.**

Explanation: The text for line II 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 (II) 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 VLCIO | VLBCM : MODULE 'module
name' NOT FOUND. A 'STEPLIB' OR A
'FORTLIB' DD IS NEEDED.**

**AFB915I CLCIO | CNREN | CXAES | CXBST :
MODULE 'module name' NOT FOUND.
A 'GLOBAL LOADLIB' IS NEEDED.**

**AFB915I VLCIO | VLBCM : ABEND 'nnn-rr'
LOADING AFBVRENA|AFBVRENC.**

**AFB915I CLCIO | CNREN | CXAES | CXBST :
ABEND 'nnn' LOADING 'module name'.**

**AFB915I VINTP : COMPOSITE MODULE 'module
name' IS FROM EARLIER RELEASE.**

Explanation: If the message is issued by CLCIO, the 'module name' needed for load mode could not be loaded. If the 'module name' is issued by other modules, the composite module AFBVRENA (under XA or ESA system) or AFBVRENC (under 370 system),

which is needed to run in load mode, could not be loaded, or is loaded but of a previous release.

For the first two forms of the message, either the statement identifying the run-time library in which the module resides is not supplied, or else the module is missing from the library.

For the next two forms of the message, the condition indicated by the abend code prevented the module from being loaded.

For the last form of the message, the composite module is from a release prior to the release the load module was linkedited with.

Supplemental Data Provided: 'module name' is the module to be loaded. **nnn** is the abend code from the CMS or MVS system; and for MVS, **rr** is the reason code 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 CMS, 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 third and fourth forms of the message, correct the problem (such as insufficient region size) indicated by the abend. If the problem reoccurs, report the problem to the people at your installation who give system support for VS FORTRAN Version 2.

For the fifth form of the message, make sure that the library is at least the same level as the one the load module linkedited with. Refer the problem to the people at your installation who give system support for VS FORTRAN Version 2. Some of the possible locations to check:

- under CMS:
 - Shared segments or DCSS.
 - GLOBAL LOADLIB command.
- under MVS:
 - Load modules installed in link-pack area (LPA).
 - The library pointed to by the JOBLIB, STEPLIB or FORTLIB DD statement.
- under TSO: the library pointed to by the FORTLIB ALLOCATE command.

AFB916I VPARM : THE AUTOTASK KEYWORD IS NOT VALID ON THIS SYSTEM.

AFB916I VPARM : THE DEBUNIT KEYWORD IS NOT VALID ON THIS SYSTEM.

Explanation: The AUTOTASK or DEBUNIT keywords were specified as a run-time parameter. These keywords request initialization of the multitasking facility. The multitasking facility is only supported on MVS systems, but it was requested when running under CMS.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Remove the AUTOTASK or DEBUNIT keyword as a run-time parameter or run the application program on an MVS system.

AFB917I VMPRM : THE AUTOTASK KEYWORD SUBPARAMETER STRING IS MISSING.

AFB917I VMPRM | VPARM : THE AUTOTASK| PARALLEL|DEBUNIT KEYWORD SUBPARAMETER STRING IS INVALID.

Explanation: The subparameters of the AUTOTASK|PARALLEL|DEBUNIT keyword in the PARM parameter of the EXEC statement were either missing or in an incorrect format.

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.

For PARALLEL, correct *numprocs* specified to be within the range 1 to 99, inclusive. The required format for the PARALLEL keyword is:

PARALLEL[(*numprocs*)]

where *numprocs* is the number of virtual processors to be used for parallel processing.

For DEBUNIT, correct the subparameters and rerun the job. The required format for the DEBUNIT keyword is:

DEBUNIT(*s1*, (*s2*,...))

where the list of units may consist of a single Fortran unit number and/or a range of unit numbers (*x*, or *yy-zz*).

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.

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 VOCMP | VMEXP : CALL TO MULTITASKING FACILITY FUNCTION ffffff WAS 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, SYNCRO, NTASKS, or SHRCOM.

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—Either:

- Specify the AUTOTASK keyword in the PARM parameter of your EXEC statement, which invokes your program, or
- Convert the program to use parallel processing constructs.

Format 4—Run your MTF application on an MVS 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 FTERR0xx 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 FTERR0xx DD statement to determine the cause of failure. Take corrective action as required.

AFB923I name : ssssssss 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, DIRECT 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 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 XXXUOPT IS IN THE RANGE OF ERROR OPTION TABLE YYYUOPT.

Explanation: An attempt to chain the error option table of an auxiliary product with the component ID "XXX" has failed. The error option table, XXXUOPT, defines a range of error numbers for table YYYUOPT 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 (XXXUOPT and YYYUOPT).

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.

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 VMIN : THE MAIN TASK PROGRAM LOAD MODULE AND THE PARALLEL SUBROUTINE LOAD MODULE ARE NOT OPERATING WITH THE SAME RELEASE 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.

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 'sssss' : 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 subroutine created by the multitasking facility or parallel feature. Unless the multitasking facility or parallel feature 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: *sssss* is the module name. *xxxxxxx* 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 tasks and use the multitasking facility or parallel feature.

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*.

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 VLClO | VLBCM : ABEND 'nnn-rr'
LOADING #VSFTASK, NO LIBRARY
INITIALIZATION WAS DONE.**
**AFB932I CLClO | CNREN | CXAES : ABEND 'nnn'
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) from the MVS or CMS system, 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# or VFEIL# 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 CMS—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 BY THE
COMPILED PROGRAM 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 ffffffff
FAILED, ERROR CODE xxxx, INFORMATION
CODE yyyy. EXECUTION TERMINATED.**

**AFB936I VSTIO : FILE DEALLOCATION FOR ffffffff
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 System Programmer's Library: System Macros and Facilities, Volume 1*, *OS/VS2 MVS System Programming Library: Job Management*, or to *MVS/ESA System Codes* 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* or *MVS/ESA System Codes* 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
ffffff	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 yyyyyyy.

Explanation: While executing in link mode, the VS Fortran Library module yyyyyyy 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:

xxxxxxx	the name of the VS FORTRAN Library module that was not available
yyyyyyy	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.

AFB938I POUTP : THE VALUE *value* IS NOT A VALID TASK IDENTIFIER.

AFB938I POUTP : THE VALUE *value* IS NOT A VALID TASK IDENTIFIER BECAUSE IT ALREADY EXISTS.

AFB938I POUTP : THE MAXIMUM NUMBER OF ORIGINATED TASKS HAS BEEN EXCEEDED.

Explanation: For format 1 of this message, the invalid value *value* was provided for the task identifier on the ORIGINATE TASK statement.

For format 2 of this message, the value *value* was provided for the ORIGINATE TASK statement. An originated task with the same identifier currently exists.

For format 3 of this message, total number of originated tasks using the ORIGINATE ANY TASK statement has been exceeded.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: For format 1, ensure that the value of the identifier provided on the ORIGINATE TASK statement is within the range 1-9999.

For format 2, either specify an identifier on the ORIGINATE TASK statement that has not already been used, or terminate the originated task with the same identifier.

For format 3, ensure that the number of originated tasks using the ORIGINATE ANY TASK statement does not exceed 9999.

AFB939I POUTP | PSHRP : ORIGINATED TASK {Sxxxx | Uxxxx} WAS SCHEDULED BUT IS CURRENTLY ACTIVE. A WAIT MUST BE ISSUED BEFORE IT CAN BE SCHEDULED AGAIN.

AFB939I POUTP | PSHRP : ORIGINATED TASK {Sxxxx | Uxxxx} MAY NOT BE SCHEDULED BECAUSE IT WAS NOT ORIGINATED UNDER THE SAME PARALLEL TASK.

AFB939I POUTP : ORIGINATED TASK {Sxxxx | Uxxxx} MAY NOT BE TERMINATED BECAUSE IT WAS NOT ORIGINATED UNDER THE SAME PARALLEL TASK.

AFB939I PWATP : ORIGINATED TASK {Sxxxx | Uxxxx} MAY NOT BE WAITED ON

BECAUSE IT WAS NOT SCHEDULED UNDER THE SAME PARALLEL TASK.
AFB939I POUTP : ORIGINATED TASK {Sxxxx | Uxxxx} MAY NOT BE TERMINATED BECAUSE IT HAS OUTSTANDING WORK ASSIGNED.

Explanation: For format 1 of this message, the same originated task identified by xxxx was scheduled twice without waiting for the originated task to complete execution of a subroutine.

For format 2 of this message, a SCHEDULE statement specifies an originated task identifier of xxxx which was not originated under the currently executing parallel task.

For format 3 of this message, a TERMINATE statement specifies an originated task identifier of xxxx which was not originated under the currently executing parallel task.

For format 4 of this message, a WAIT FOR TASK statement was executed under a parallel task that did not schedule the waited parallel task identified by xxxx.

For format 5 of this message, A TERMINATE TASK statement was specified for an originated task identifier of xxxx which has outstanding work assigned.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: For format 1, ensure that a WAIT FOR TASK, or WAIT FOR ALL TASKS statement is issued for completion of work that has been scheduled.

For format 2, ensure that the originated task being scheduled has been originated under the correct parallel task.

For format 3, ensure that the originated task being terminated has been originated under the correct parallel task.

For format 4, ensure that the WAIT FOR TASK statement identifies an originated task that was previously scheduled.

For format 5, ensure that a WAIT FOR statement is issued before the originated task is terminated.

AFB940I POUTP : NO UNSCHEDULED ORIGINATED TASKS ARE AVAILABLE. THE SCHEDULE ANY TASK STATEMENT CANNOT BE PROCESSED.

AFB940I PWATP : NO SCHEDULED ORIGINATED TASKS ARE OUTSTANDING. THE WAIT FOR ANY TASK STATEMENT CANNOT BE PROCESSED.

Explanation: For format 1 of this message, all parallel tasks that have been originated by the scheduling routine are currently scheduled.

For format 2 of this message, a WAIT FOR ANY TASK was issued when there was no parallel task scheduled.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: For format 1, ensure that a sufficient number of tasks have been originated or that previously scheduled tasks have been waited for.

For format 2, ensure that the WAIT FOR ANY TASK statement is issued only when there is an outstanding parallel task scheduled.

AFB941I PSHRP : COMMON BLOCK *name* IS NOT AVAILABLE IN PARALLEL TASK {Sxxxx|Uxxxx} FOR SHARING OR COPYING.

Explanation: A SCHEDULE statement with a SHARING, COPYING, COPYINGI, or COPYINGO clause for common block *name* is encountered, without the common block being available.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Ensure that the common block has been declared by the scheduling routine.

AFB942I PLOKP : THE LOCK *////* MAY NOT BE OBTAINED BECAUSE IT HAS ALREADY BEEN OBTAINED.

AFB942I PLOKP : THE LOCK *////* MAY NOT BE { OBTAINED | RELEASED } BECAUSE IT HAS NOT BEEN ORIGINATED OR HAS BEEN TERMINATED.

AFB942I PEVTP : THE EVENT *eeee* MAY NOT BE { POSTED | WAITED ON } BECAUSE IT HAS NOT BEEN ORIGINATED OR HAS BEEN TERMINATED.

Explanation: For format 1 of this message, the PLLOCK service routine or PLCOND function attempted to obtain a lock that the same parallel thread already has obtained.

For format 2 of this message, PLLOCK, PLCOND, or PLFREE was passed lock identifier *////* that has not been created by the PLORIG service routine, or has been terminated.

For format 3 of this message, the PEPOST or PEWAIT service routine was passed event identifier *eeee* that has not been created by the PEORIG service routine, or has been terminated.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: For format 1, ensure that a lock is not obtained a second time by the same parallel thread.

For format 2, ensure that the PLORIG service routine is used to originate the lock and that the PLTERM service routine has not been used to terminate it.

For format 3, ensure that the PEORIG service routine is used to originate the event, and that the PETERM service routine has not been used to terminate it.

AFB943I PEVTP | PLOKP : THE ARGUMENT LIST FOR *pppppp* IS IN AN INCORRECT FORMAT.

Explanation: Too many or too few arguments have been passed to a lock or event service routine or function.

Supplemental Data Provided: *pppppp* can be PLORIG, PLTERM, PLLOCK, PLCOND, PLFREE, PEORIG, PEWAIT, PEPOST, or PETERM.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Ensure the proper number of arguments are passed to the lock or event service routine.

AFB944I PEVTP : AN EVENT IS NOT ORIGINATED BECAUSE THE ARGUMENTS HAVE INCORRECT VALUES.

Explanation: Incorrect values were passed to the PEORIG service routine.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Ensure that the post count, wait count, and unique indicators have correct values.

AFB945I PEVTP : AN EVENT MAY NOT BE ORIGINATED WITHIN PARALLEL CALL, PARALLEL LOOP, OR PARALLEL SECTION.

AFB945I PEVTP : AN EVENT MAY NOT BE TERMINATED WITHIN PARALLEL CALL, PARALLEL LOOP, OR PARALLEL SECTION.

AFB945I PEVTP : AN EVENT MAY NOT BE POSTED WITHIN PARALLEL CALL, PARALLEL LOOP, OR PARALLEL SECTION.

AFB945I PEVTP : AN EVENT MAY NOT BE WAITED FOR WITHIN PARALLEL CALL, PARALLEL LOOP, OR PARALLEL SECTION.

Explanation: For format 1 of this message, a CALL PEORIG statement is coded in a parallel loop or parallel section language construct, or in the subroutine invoked by a parallel call statement; or a subroutine or function called in the loop, the section, or the subroutine has a CALL PEORIG statement.

For format 2 of this message, a CALL PETERM statement is coded in a parallel loop or parallel section language construct, or in the subroutine invoked by a parallel call statement; or a subroutine or a function called in the loop, the section, or the subroutine has a CALL PETERM statement.

For format 3 of this message, a CALL PEPOST statement is coded in a parallel loop or parallel section language construct, or in the subroutine invoked by a parallel call statement; or a subroutine or function called in the loop, the section, or the subroutine has a CALL PEPOST statement.

For format 4 of this message, a CALL PEWAIT statement is coded in a parallel loop or parallel section language construct, or in the subroutine invoked by a parallel call statement; or a subroutine or a function called in the loop, the section, or the subroutine has a CALL PEWAIT statement.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: For format 1, remove the CALL PEORIG statement.

For format 2, remove the CALL PETERM statement.

For format 3, remove the CALL PEPOST statement.

For format 4, remove the CALL PEWAIT statement.

AFB946I VINTP : A STOP STATEMENT OR CALL TO EXIT OCCURRED IN A PARALLEL THREAD.

Explanation: A STOP statement or call to the EXIT or SYSRCX service routine occurred in a parallel loop, parallel section, parallel call, or scheduled subroutine.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Remove the STOP statement or call to the EXIT or SYSRCX service routines from the parallel loop, parallel section, parallel call, or scheduled subroutine.

AFB947I PDXSP : A DEADLOCK SITUATION HAS OCCURRED.

Explanation: A condition was detected where all of the virtual processors would be put into a wait state with no more parallel work that can be executed, yet execution of all work is not complete.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Review all use of lock and event services, and correct any deadlocks that are occurring.

AFB948I POUTP : INSUFFICIENT STORAGE AVAILABLE TO ORIGINATE A TASK.

AFB948I PWATP | PSHRP : INSUFFICIENT STORAGE AVAILABLE TO SCHEDULE A TASK.

AFB948I PINLP : INSUFFICIENT STORAGE AVAILABLE FOR PARALLEL LOOP OR PARALLEL SECTIONS.

AFB948I PCALP : INSUFFICIENT STORAGE AVAILABLE FOR PARALLEL CALL.

AFB948I PLOKP : INSUFFICIENT STORAGE AVAILABLE TO ORIGINATE A LOCK.

AFB948I PEVTP : INSUFFICIENT STORAGE AVAILABLE TO ORIGINATE AN EVENT.

AFB948I PXNIT : INSUFFICIENT STORAGE FOR PARALLEL INITIALIZATION.

AFB948I PLOKP : INSUFFICIENT STORAGE AVAILABLE TO OBTAIN LOCK *////*.

AFB948I PEVTP : INSUFFICIENT STORAGE AVAILABLE TO POST EVENT *eeee*.

Explanation: There was not enough virtual storage available to process the statement, library service routine, or initialize the parallel processing environment.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Either increase the virtual storage available to the program or reduce the amount of parallel work being generated.

AFB949I VINTP : A PARALLEL FUNCTION CANNOT BE PROCESSED BECAUSE MTF IS ACTIVE.

Explanation: Parallel language statements, parallel code generated by automatic parallelization, or lock or event services were encountered, and the environment for parallel processing could not be initialized because MTF is active.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Either remove the AUTOTASK keyword, and ensure the DSPTCH, SHRCOM, or SYNCRO routines are not called, or ensure parallel language constructs, automatically parallelized code, or lock or event library services are not executed when the MTF environment is active.

AFB950I VPARAM : INVALID NUMBER OF VIRTUAL PROCESSORS SPECIFIED.

AFB950I PXNIT : THE NUMBER OF VIRTUAL PROCESSORS SPECIFIED COULD NOT BE OBTAINED.

Explanation: For format 1 of this message, an invalid number of virtual processors was specified on the PARALLEL run-time option.

For format 2 of this message, the number of virtual processors specified on the PARALLEL run-time option could not be obtained.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: For format 1, ensure the number of virtual processors specified is within the correct range, and that the PARALLEL run-time option is correctly specified.

For format 2, on VM ensure that the number of virtual processors specified is less than the number of virtual CPUs that can be defined. On MVS, ensure there is enough storage available to attach the number of tasks for the virtual processors specified on the PARALLEL run-time option.

AFB951I POUTP | PWATP | PSHRP : THE *stmt* STATEMENT OCCURS WITHIN A PARALLEL LOOP, PARALLEL SECTION, OR PARALLEL CALL.

AFB951I PCALP : THE PARALLEL CALL STATEMENT OCCURS WITHIN A PARALLEL LOOP OR PARALLEL SECTION.

Explanation: For format 1 of this message, an ORIGINATE, SCHEDULE, TERMINATE, or WAIT FOR statement indicated by *stmt* occurred within a subroutine or a function given control in a parallel loop, parallel section, or parallel call.

For format 2 of this message, a PARALLEL CALL statement occurs within a subroutine or a function given control in a parallel loop or section.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Remove the statement from any subroutine called within the parallel loop, parallel section, or parallel call.

AFB952I PXEXT : PARALLEL PROCESSING CANNOT BE USED ON THIS SYSTEM.

AFB952I PXEXT : MODULE BPPARFOR IS NOT AVAILABLE FOR PARALLEL PROCESSING SUPPORT.

AFB952I PXEXT : INVALID NUMBER OF VIRTUAL PROCESSORS SPECIFIED.

AFB952I PXEXT : PARALLEL FORTRAN INTERFACE INITIALIZATION FAILED.

AFB952I PXMOD : MODULE *mmmmmmmm* NOT EXECUTED FROM A LOADLIB OR RELOCATABLE MODULE.

AFB952I PXEXT : MODULE *mmmmmmmm* NOT FOUND FOR PARALLEL EXECUTION.

AFB952I PXEXT : PARALLEL PROGRAM LOAD MODULE NOT FOUND.

AFB952I name: AUTHORIZED PROGRAM EXECUTION NOT SUPPORTED.

Explanation: For format 1 of this message, parallel processing is not supported on the system the program is being run on.

For format 2 of this message, the CMS PRPQ To Support Parallel Processing In VS FORTRAN (5799-DGW) is not available.

For format 3 of this message, the number of virtual processors requested could not be obtained.

For format 4 of this message, initialization failed during the execution of the CMS PRPQ To Support Parallel Processing in VS FORTRAN.

For format 5 of this message, the module *mmmmmmmm* was not relocatable when it was created with the CMS GENMOD command, or was not contained in a CMS LOADLIB.

For format 6 of this message, the module *mmmmmmmm* was not found during parallel processing.

For format 7 of this message, the program being executed could not be found in storage.

For format 8 of this message, authorized program execution was detected.

Supplemental Data Provided:

name PXEXT, VASYP, VINTP

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response:

For format 1, ensure the parallel program is run on a system that is supported.

For format 2, ensure that the CMS PRPQ To Support Parallel Processing In VS FORTRAN (5799-DGW) is installed.

For format 3, ensure that the number of virtual processors requested on the PARALLEL run-time option is less than or equal to the number of virtual CPUs that are allowed to be defined.

For format 4, review the messages that precede this message.

For format 5, ensure that a module is created using the GENMOD command and that the RLDSAVE command is used when using the LOAD and INCLUDE commands, or ensure that the module is a member of a LOADLIB.

For format 6, ensure that the TSO LOADGO command or loader is not being used to invoke the parallel program.

For format 7, ensure that assembler routines which invoke the loader are not used in a parallel program.

For format 8, ensure that the VS FORTRAN program is not link-edited or invoked by a module that is link-edited with the AC(1) parameter. For a parallel program, ensure that the program does not reside in a library that contains programs protected by the RACF PROGRAM class or in an APF authorized library.

AFB953I VINTP : DEBUG RUN-TIME OPTION SPECIFIED WITH MORE THAN ONE VIRTUAL PROCESSOR.

Explanation: Only PARALLEL run-time option with 1 virtual processor can be specified with DEBUG run-time option.

Standard Corrective Action: Execution terminates with return code of 16.

Programmer Response: Specify PARALLEL(1) run-time option, or remove DEBUG run-time option.

AFB960I VOCMP : DATA SPACE ALLOCATION FAILED DURING "ffffff" PROCESSING, RETURN CODE = ccc, REASON CODE = rrr

Explanation: An unexpected failure occurred in an MVS/ESA or VM/ESA system function while attempting to allocate the storage to contain an extended common block.

Supplemental Data Provided: fffffff identifies the failing system function. ccc identifies the return code. rrr identifies the reason code.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Consult your systems programmer for an explanation and analysis of the failure.

AFB961I VOCMIP : COMMON BLOCK "cccccccc" EXCEEDS THE MAXIMUM SIZE FOR AN EXTENDED COMMON BLOCK.

Explanation: The operating system has reported that the identified extended common block cannot be allocated because it exceeds the size of an ESA/390* data space. The size of these blocks may not exceed 2G bytes.

Supplemental Data Provided: cccccccc is the name of the common block.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Reduce the size of the extended common block.

* ESA/390 is a trademark of the International Business Machines Corporation.

AFB962I VOCMP : MUST BE USING 31-BIT ADDRESSING MODE TO OBTAIN STORAGE FOR COMMON "cccccccc".

Explanation: Use of extended common blocks requires that the VS FORTRAN program execute in 31-bit addressing mode (AMODE 31). The program attempting to use an extended common block was executing in 24-bit addressing mode (AMODE 24).

Supplemental Data Provided: cccccccc is the name of the common block.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: The user should recreate the program load module using the AMODE 31 option.

AFB963I VOCMP : DATA SPACE CREATION ERROR DURING "ffffff" PROCESSING, RETURN CODE = ccc, REASON CODE = rrr.

Explanation: An unexpected failure occurred in an MVS/ESA or VM/ESA system function while attempting to allocate the storage to contain an extended common block.

Supplemental Data Provided: fffffff identifies the failing system function. ccc identifies the return code. rrr identifies the reason code.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Consult your systems programmer for an explanation and analysis of the failure.

AFB964I VOCMP : DATA SPACE EXTEND FAILED DURING "ffffff" PROCESSING, RETURN CODE = ccc, REASON CODE = rrr.

Explanation: An unexpected failure occurred in an MVS/ESA or VM/ESA system function while attempting to allocate the storage to contain an extended common block.

Supplemental Data Provided: fffffff identifies the failing system function. ccc identifies the return code. rrr identifies the reason code.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Consult your systems programmer for an explanation and analysis of the failure.

AFB965I VINTP : EXTENDED COMMON BLOCKS CANNOT BE USED TOGETHER WITH THE INTERACTIVE DEBUG FACILITY. REMOVE THE "DEBUG" OPTION AND RETRY THE PROGRAM.

AFB965I VINTP : EXTENDED COMMON BLOCKS CANNOT BE USED TOGETHER WITH THE MULTITASKING FACILITY. REMOVE THE "AUTOTASK" OPTION AND RETRY THE PROGRAM.

Explanation: A program containing extended common blocks cannot be run when either the interactive debug facility or the multitasking facility is used.

Supplemental Data Provided: xxxxxxxx is the name of an extended common block.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Remove the DEBUG and/or AUTOTASK execution option to run the program.

AFB966I VOCMP : EXTENDED COMMON BLOCK "xxxxxxx" COULD NOT BE ALLOCATED BECAUSE THE NUMBER OF DATA SPACES REQUIRED EXCEEDED THE LIMITS IMPOSED BY YOUR INSTALLATION.

Explanation: Extended common block xxxxxxxx was not allocated because the required data space could not be obtained. This occurred because the installation would not permit additional data spaces to be allocated. There are two reasons for this: (a) the maximum number of data spaces has been obtained, or (b) the total amount of storage for all data spaces has been allocated. (This message occurs for MVS only.)

AFB966I VOCMP : EXTENDED COMMON BLOCK "xxxxxxx" COULD NOT BE ALLOCATED BECAUSE THE MAXIMUM NUMBER OF DATA SPACES YOU ARE ALLOWED TO USE HAS BEEN EXCEEDED.

AFB966I VOCMP : EXTENDED COMMON BLOCK "xxxxxxx" COULD NOT BE ALLOCATED BECAUSE THE TOTAL AMOUNT OF STORAGE FOR ALL DATA SPACES AVAILABLE TO YOU HAS BEEN EXCEEDED.

Explanation: Extended common block xxxxxxxx was not allocated because the required data space could not be obtained. This occurred because the XCONFIG statement in the directory does not allow additional data spaces to be obtained.

Programmer Response: Consult your systems programmer to increase the values for the XCONFIG statement. This will allow additional data spaces to be allocated. (This message occurs for CMS only.)

AFB970I VABEX : * ESA/390 ENVIRONMENT
ERROR DETECTED *****
**AFB971I VABEX : MODULE xxxxxxxx MUST BE
COMPILED WITH THE "EMODE" COM-
PILER OPTION.**

Explanation: Abnormal program termination occurred because an attempt was made to call a subroutine or function not prepared to handle extended common blocks.

Message AFB970I identifies the reason for the error. Specifically, a program using extended common blocks attempted to call a subroutine or function not compiled to handle these common blocks.

Message AFB971I gives the name of the subroutine or function that was called. This subroutine or function must be compiled using the EMODE compile-time option.

Supplemental Data Provided: xxxxxxxx identifies the subroutine or function causing the abnormal program termination condition.

Standard Corrective Action: Execution terminates with message AFB240I.

Programmer Response: Recompile module xxxxxxxx with the EMODE compile-time option.

Index

Special Characters

`_` (underscore) 6
`,` (comma) 6
`=` (equal sign) 6
`:` (colon) 6
`:` format code 126
`!` (exclamation point) 10
`/` (slash) 6
`/` format code 125
`/=` (not equal to) 39
`.` (period) xiii, 6
`'` (apostrophe) 6
`"` (quotation mark) 6, 9
`-` (minus sign) 6
(blank) 6
`@PROCESS` statement 52
`$` (currency symbol) 6
`$` format code 126
`*` (asterisk) 11, 28
`+` (plus sign) 6

Numerics

66-level language 405
77-level language 405

A

A format code 120
abend data, post 436
abnormal termination subroutines
 SYSABN and SYSABD 326
ABS intrinsic function 284
ABS/IABS error message 481
absolute value functions
 ABS/IABS/DABS/QABS 295
 CABS 295
 CDABS 295
 CQABS 295
ACOS intrinsic function 282
ACTION specifier, error messages 448
action statement
 definition 3
actual argument
 array name 28
 in a function 133
 in a subroutine 230
 in an ENTRY statement 96
adjustable dimension declarator 29
AFB library messages 438
AIMAG intrinsic function 285

AINTR intrinsic function 283
ALGAMA 300
ALLOCATE statement 54
ALLOCATED intrinsic function 287
allocation
 statements 51
allocation of a file, dynamic 148, 169
ALOG/ALOG10
 error message 480
alphabetic primary
 See primary
alternate return parameter 96
alternative mathematical library routines
 documentation 317
American National Standard (ANS) 405
ANINTR intrinsic function 283
apostrophe 6
 with character constant 19
ARCOS 296
arcsine and arccosine functions
 ACOS 296
 ASIN 295
 DACOS 296
 DASIN 296
 QARSIN/QARCOS 297
arctangent functions
 ATAN 297
 ATAN2 297
 DATAN 297
 DATAN2 298
 QATAN/QATAN2 298
ARGSTR routine 342
argument
 actual
 in ENTRY statement 96
 assembler language 414
 dummy
 in ENTRY statement 96
 vector registers 418
argument vector registers 418
arithmetic
 assignment statement, conversion rules
 arithmetic 58
 complex 58
 integer 58
 real 58
 constant
 complex 17
 constant 14
 integer 14
 primary 33
 real 15

- arithmetic (*continued*)
 - expression 407
 - examples 37
 - rules for constructing 33
 - type and length of 36
 - use of parentheses in 35
 - IF statement 138
 - operation
 - addition 33
 - division 33
 - evaluation of functions 34
 - exponentiation 33
 - first operand 35
 - multiplication 33
 - order of computation 34
 - subtraction 33
 - unary 33
 - operator
 - in logical expression 43
 - kinds of 33
- array
 - actual argument 27, 28
 - assumed-size 28
 - character 30
 - declarator 27
 - definition 26
 - dimension error message 467
 - DIMENSION statement 85
 - dimensions 27, 85
 - dummy argument 28
 - dynamic dimensions 29
 - elements 26
 - examples 27
 - in parallel loop 270
 - items, sample program and output 431
 - output format 425
 - pointee 28
 - size and type declaration 27
 - subscripts 26
 - upper dimension bound as asterisk 28
- ARSIN 295
- ASIN intrinsic function 282
- assembler language
 - calling sequence 413
- assign a name to a main program 182
- ASSIGN statement 55
- assigned GO TO statement 136
- ASSIGNM routine
 - examples 341
 - rules 341
- assignment statement
 - character 60
 - logical 61
- associating in data-in-virtual 366, 369
- assumed-size array 28

- asterisk 6
- asynchronous
 - execution 265
 - I/O 406
 - I/O error messages 463, 468
 - READ statement 183
 - WRITE statement 237
- AT statement
 - debugging 62
 - in debug packet 81
- ATAN intrinsic function 282
- ATAN/ATAN2 error message 481
- ATAN2 intrinsic function 282
- automatic
 - function selection 407
 - in equivalence 97
- AUTOMATIC statement 63
- AUTOTASK
 - keyword 401

B

- B format code 115
- BACKSPACE statement 63
- basic real constant 15
- binary
 - constant 20
 - data transmission 115
- bit function error message 462
- bit functions, explicitly called 417
- bit manipulation
 - assembler information 417
 - routines 319
 - subroutine 343
- bit-manipulation intrinsic functions 288
- blank
 - format code 124
 - FORMAT statement 124
- blanks, leading 108
- BLOCK DATA 51, 65
- block IF statement
 - ELSE 140
 - ELSE IF 141
 - END IF 139
- BN format code 124
- boz constant 14
- BTEST intrinsic function 289
- BTSHS error message 462
- BYTE type statement 99
- BZ format code 124

C

- CALL macro instruction 413
 - for vector intrinsic functions 417

- CALL statement 67
 - ASSIGNM 340
 - CDUMP/CPDUMP 322
 - CLOCK 336
 - CLOCKX 336
 - CPUTIME 337
 - DATIM 339
 - DATIMX 339
 - DIVCML 371
 - DIVINF 364
 - DIVINV 368
 - DIVRES 373
 - DIVSAV 372
 - DIVTRF 367
 - DIVTRV 372
 - DIVVWF 366
 - DIVVWV 369
 - DUMP/PDUMP 321
 - DVCHK 319
 - ERRMON 383
 - ERRSAV 384
 - ERRSET 385
 - ERRSTR 388
 - ERRTRA 388
 - EXIT 326
 - FILEINF 327
 - IGNFHDD 334
 - IGNFHU 334
 - MVBITS 343
 - OVERFL 319
 - SDUMP 323
 - subroutines, varying-view 368
 - SYSABD 326
 - SYSABN 326
 - SYSRCS 324
 - SYSRCT 324
 - SYSRCX 324
 - UNTANY 331
 - UNTNOFD 331
- calling routines in assembler language 413
- carriage control 106
- CCMPR# 313
 - error message 468
- CDDVD#/CDMPY# 310
- CDSIN/CDCOS error message 486
- CDSQRT
 - effect of an argument error 306
- CDVD#/CMPY# 310
- CEXP/CDEXP error messages 484, 486
- change options 385
- CHAR intrinsic function 288
- CHAR specifier
 - on INQUIRE statement 153
 - on OPEN statement 172
- character
 - array element
 - READ statement, direct access, formatted 187
- character (*continued*)
 - array name
 - READ statement, direct access, formatted 187
 - assignment statement 60
 - constant
 - definition 19
 - length of 19
 - mixed 19
 - mixed, invalid 31
 - mixed, substring 31
 - READ statement, direct access, formatted 186
 - transmission 121
 - uses of 19
 - data dump 322
 - data transmission 120
 - double-byte 6
 - expression
 - operators 38
 - READ statement, direct access, formatted 187
 - use of parentheses in 39
 - manipulation routines, assembler information 417
 - operators in logical expressions 43
 - set, collating sequence 4
 - skipping 122
 - source statement 4
 - special 4
 - substring 30
 - variable
 - READ statement, direct access, formatted 187
 - storage length 24
 - substring 30
- CHARACTER type statement 99
- CHARLEN compile-time option 102
- CI compile-time option 145
- CITFN error message 467
- CLCI0 error message 492
- CLEXP error message 486
- CLLOG error message 486
- CLOCK routine 336
- CLOCKX routine 336
- CLOG/CDLOG error messages 484, 486
- CLOSE statement 71
 - error messages 464, 467
- CLSCN error messages 486
- CMOVE error messages
 - source length for character move invalid 468
 - target length for character move invalid 468
- CMOVE# 313
 - error message 468
- CMPLX intrinsic function 287
- CMPY# 310
- CMS considerations
 - operating system support xvi
- CNCAT# 313
 - error message 469

- coding your program
 - user exit routine 386
- colon 6
- colon format code 126
- comma 6
- comma in formatted I/O 108
- comments
 - definition 9
 - fixed-form 11
 - free-form 9
 - valid placement 10
- common block
 - blank 75
 - extended
 - asynchronous READ statement and 185
 - asynchronous WRITE statement and 237
 - MTF and 401
 - WAIT statement and 236
 - named 75
 - parallel program, using in 74
 - to initialize variables 66
- COMMON statement 73, 76
- compare complex numbers 310
- compile-time
 - serializations, parallel 274
- compiler directives
 - EJECT statement 51, 91
 - INCLUDE statement 51, 145
- complex constant 17
 - examples 17
- complex data type 406
- COMPLEX format 321, 424
- complex multiply and divide functions
 - CDDVD#/CDMPY# 315
 - CDVD#/CMPY# 315
 - CQMPY#/CQDVD# 315
- complex multiply and divide routines
 - CDDVD#/CDMPY# 310
 - CDVD#/CMPY# 310
 - CQMPY#/CQDVD# 310
- COMPLEX type statement 99
- complex variable, storage length 24
- complex-to-integer 310
- computed GO TO statement 137
- concatenation
 - operation 39
 - with double-byte characters 39
- concatenation operand error message 469
- CONJG intrinsic function 285
- connected file
 - BACKSPACE statement 64
 - determining properties of a 147
 - ENDFILE statement 93
 - general description 49
 - INQUIRE statement 147
 - OPEN statement 168
- connected file (*continued*)
 - preconnection 50
 - READ statement
 - formatted with direct access 188
 - formatted with keyed access 191
 - list-directed with external devices 201
 - NAMELIST with external devices 205
 - unformatted with keyed access 213
 - unformatted with sequential access 216
 - REWIND statement 219
 - REWRITE statement 220
 - WRITE statement
 - formatted with direct access 240
 - formatted with keyed access 242
 - formatted with sequential access 245
 - list-directed with external devices 251
 - NAMELIST with external devices 255
 - unformatted with direct access 259
 - unformatted with keyed access 261
 - unformatted with sequential access 263
- connected unit
 - See *also* connected file
 - general description 49
- connecting files and units
 - file, determining 147
 - unit, determining 147
- connection, file/unit, general description 49
- constant
 - arithmetic 14
 - assign a name to 176
 - binary 20
 - boz 14
 - character 19
 - classes 14
 - complex 17
 - definition 14
 - expression 39
 - hexadecimal 21
 - Hollerith 20
 - integer 14
 - logical 18
 - mixed character 19
 - name, assigning 176
 - octal 20
 - real 15
 - typeless 14
- construct 264
- continuation line
 - fixed-form 12
 - free-form 10
- CONTINUE statement 76
- continued line, free-form 10
- control
 - character 106
 - RETURN statement 217
 - statement
 - assigned GO TO 136

- control (*continued*)
 - statement (*continued*)
 - CALL 67
 - computed GO TO 137
 - CONTINUE 76
 - DO 86
 - DO WHILE 86
 - ELSE 140
 - ELSE IF 141
 - END 91
 - END DO 92
 - ENDIF 139
 - execution 47
 - GO TO 135
 - IF 138
 - PAUSE 178
 - RETURN 217
 - STOP 229
 - unconditional GO TO 137
- control flow information 436
 - output format 426
- conversion
 - rules 58
 - data 106
- corrective action
 - after error 392
 - after mathematical subroutine error 397
 - after program interrupt 396
- COS
 - error message 481
- COS intrinsic function 282
- COSH intrinsic function 283
- cosine functions 303
- COTAN intrinsic function 282
- cotangent functions 306
- CPUTIME routine 337
- CQEXP
 - error messages 485
- CQLOG
 - error message 485
- CQMPY#/CQDVD# 310
- CQSCN error messages 485
- CQSIN/CQCOS error message 485
- CQSQRT
 - CQSQRT 306
- create a file/preconnected file 168
- CSEXP error messages 484
- CSIN/CCOS error message 484
- CSLOG error message 484
- CSSCN error messages 484
- CSTIO error messages 497
- currency symbol 6
- CVIOS error messages
 - end of data set 472
 - ENDFILE statement 463
 - file positioning I/O statement in direct access mode 463

- CVIOS error messages (*continued*)
 - I/O request made to unopened VSAM file 464
 - I/O statement 464
 - key sequenced data set 463
 - OPEN statement ACTION specifier conflict 453
 - record addition conflict 463
 - record length too long 463
 - record number 476
 - unit number out of range 474
- CXMPR# 310

D

- D format code 111
- DABS 295
- DARCOS 296
- DARSIN 296
- data
 - conversion rules 106
 - definition 14
 - double-byte, moving 340
 - kinds of 14
 - transfer 122
 - type
 - character 406
 - complex 406
 - double precision 409
 - extended precision 409
 - integer 408
 - logical 408
 - real 409
 - types and lengths 23
- DATA statement 48, 76
 - character data in 77
 - implied DO in 89
- data-in-virtual
 - subroutines 361
 - error messages 458, 461
 - page-alignment requirements 381
 - using with parallel programs 362
- date
 - routines
 - DATIM 339
 - DATIMX 339
- DBCS
 - See double-byte character set
- DBLE intrinsic function 286
- DCONJG intrinsic function 285
- DCOTAN
 - error message 483
- DDIM option 29
- ddnames
 - default naming conventions 151
 - for named/unnamed files 49
- DEALLOCATE statement 78

- DEBUG statement 48
 - AT statement 62, 81
 - DISPLAY statement 81, 86
 - END DEBUG statement 81, 92
 - examples 81
 - SUBCHK function 81
 - TRACE OFF statement 81, 233
 - TRACE ON statement 81, 234
- debug statements 79
- debug, considerations when using 81
- debugging
 - starting 62
- decimal point in format codes 108
- default
 - ddname conventions 151
 - options 392
- define values of
 - array elements
 - DATA statement 76
 - EXPLICIT type statement 102
 - substrings 76
 - variables
 - DATA statement 76
 - EXPLICIT type statement 101
- DELETE specified on CLOSE statement 72
- DELETE statement 84
- deleting records 84
- delimiter type
 - INQUIRE statement 153
- DERFC 299
- DEXP
 - error message 482
- DGAMMA/DLGAMA
 - error message 489
- digit 6
- DIM intrinsic function 285
- DIMAG intrinsic function 285
- dimension boundary specified as asterisk 28
- DIMENSION statement 85
 - array declaration 28
 - examples 28
- dimensions, object-time 29
- DIOCS error messages, unit number out of range 474
- DIOCS/VDIOS error message 476
- direct access
 - error messages 441, 476
 - files 173
 - I/O 406
 - INQUIRE statement 154
 - READ statement
 - formatted 186
 - unformatted 209
 - WRITE statement
 - formatted 238
 - unformatted 257
- directives
 - compiler 51
- disconnecting
 - external file 71
- DISPLAY statement 86
 - in debug packet 81
- DIV facility
 - See data-in-virtual
- DIVCML subroutine 371
- divide complex numbers 310
- divide-check
 - See *also* DVCHK
 - routine
 - error message 441
- divide-check exception test 319
- DIVINF subroutine 364
- DIVINV subroutine 368
- DIVRES subroutine 373
- DIVSAV subroutine 372
- DIVTRF subroutine 367
- DIVTRV subroutine 372
- DIVVWF subroutine 366
- DIVVWV subroutine 369
- DLOG/DLOG10
 - error message 482
- DO loop 86
 - See *also* range of a DO loop
- DO statement
 - labeled 86
 - unlabeled 86
- DO WHILE statement
 - labeled 86
 - unlabeled 86
- DOAFTER statement 272
- DOBEFORE statement 271
- documentation of IBM extensions xv
- DOEVERY statement 272
- dollar format code 126
- DOUBLE COMPLEX type statement 99
- double precision
 - data editing 111
 - data type 409
 - storage length 24
 - type 99, 143
- DOUBLE PRECISION type statement 99
- double-byte character data
 - documentation xiii
 - moving strings 340
- double-byte character set 12
 - character constant 19
 - compile-time option 12
 - in continued source lines 10, 12
 - names 9, 25
 - representation 6
 - support 172
 - with INQUIRE 153

double-byte character set (*continued*)
with OPEN 172

DP

- assign 60
- extend 60
- float 60

DPROD intrinsic function 285

drop sign 60

DSINH/DCOSH error message 483

DSPAN#/DSPN2# 314

- assembler language requirements 417

DSPN4# 314

DSPN5# 314

DSPTCH subroutine 402

DSQRT

- error message 482

DTAN

- error message 483

dummy argument

- array name 28

- in a function 133

- in a subroutine 231

- in an ENTRY statement 96

dump

- an area of storage 322

- storage 321

- symbolic 323

DUMP/PDUMP routine 321

- format specifications 321

- output 321

- programming considerations 321

DVCHK routine 319

DYCMN# 314

dynamic file allocation 49

- error messages 497

- with FILEINF routine 327

- with INQUIRE statement 148

- with OPEN statement 169

dynamically dimensioned 29, 270

E

E format code 111

EBCDIC

- double-byte character 6, 9

- names 9

ECCMN# 314

effect of an argument error 292

- CABS 295

- notation 293

EJECT statement 51, 91

elements, major language 405

ELSE IF block 141

ELSE IF statement 141

ELSE statement 140

END DEBUG statement 92

- in debug packet 81

END DO Statement 92

- in a DO loop 92

end execution error message 491

END IF statement 139

end of data set error message 472

end page 106

END SECTIONS statement 276

END statement 91

- in a function 92

- in a subroutine 92

end-of-line comment 11

ENDFILE statement 92

- error message 463

enhancements to product xvi

ENTRY statement 94

equal sign 6

EQUIVALENCE statement 97

ERR parameters honored for I/O errors 392

ERRMON subroutine 383

error

- corrective action after 392

- handling subroutines 320, 382

- message unit 464

- messages 441

 - execution 441

 - library 438

 - operator 439

 - program interrupt 438

- monitor routine 383

- option table 382

error functions 298

- DERF/DERFC 299

- ERF/ERFC 298

- QERF/QERFC 299

ERRSAV subroutine 384

ERRSET subroutine 385

ERRSTR subroutine 388

ERRTRA subroutine 388

event cycle parameters

- postcount 345

- unique 345

- waitcount 345

event routines

- PEORIG routine 346

- PEPOST routine 348

- PETERM routine 348

- PEWAIT routine 348

exclamation point 10

executable program

- definition 3

- names 8

executable statement, definition 3

execution

- environment, initialization 413

- execution (*continued*)
 - error message 441
 - termination 326
- EXIT routine 326
- EXIT statement 273
- EXP
 - error message 480
- EXP intrinsic function 281
- explicit type statement 99
- explicitly called functions
 - absolute value 295
 - arcsine and arccosine 295
 - arctangent 297
 - error function 298
 - exponential 299
 - gamma and log gamma 300
 - hyperbolic sine and cosine 301
 - hyperbolic tangent 301
 - logarithmic 302
 - sine and cosine 303
 - square root 305
 - tangent and cotangent 306
- explicitly called routines
 - bit function 417
- exponent
 - See *also* OVERFL routine
 - overflow exception 319
 - real 16
 - testing 319
- exponential functions
 - explicitly called 299
 - CEXP/CDEXP 299
 - CQEXP 300
 - DEXP 299
 - EXP 299
 - QEXP 300
 - implicitly called 316
 - FCD CD# 317
 - FCDXI# 316
 - FCQCQ# 317
 - FCQXI# 316
 - FCXPC# 317
 - FCXPI# 316
 - FDXPD# 316
 - FDXPI# 316
 - FQXP2# 316
 - FQXPI# 316
 - FQXPQ# 316
 - FRXPI# 316
 - FRXPR# 316
- exponential routines
 - implicitly called 314
- exponentiation 314
- expressions 32, 407
 - arithmetic 33
 - character 38

- expressions (*continued*)
 - evaluation of 32
 - examples 33
 - kinds of 32
 - logical 41
 - relational 39
 - type of primary in 33
- extended
 - error handling 382
 - precision data type 409
 - precision routine results 417
- extended integer
 - value error message 491
- extensions, IBM 405
 - documentation of xv
- external file 219
 - I/O unit connected to 175
 - position 219
 - unit not connected to file 175
- external files
 - sequential 92
- external procedure definition 3
- EXTERNAL statement 103

F

- F format code 110
- F8XP8#
 - error message 489
- F8XPI#
 - error message 489
- FALSE 18
- FCD CD#
 - effect of an argument error 317
- FCD CD#, error message 479
- FCDX8#
 - error message 491
- FCDXI# 310
 - error message 479
- FCQCQ#
 - effect of an argument error 317
- FCQCQ#, error message 483
- FCQX8#
 - error message 491
- FCQXI# 310
 - effect of an argument error 316
 - error message 483
- FCXP8#
 - error message 490
- FCXPC#
 - effect of an argument error 317
 - error estimate 317
- FCXPC#, error message 479
- FCXPI# 310
 - error message 479

- FDXP8#
 - error message 490
- FDXPD# 310
 - accuracy explanation 316
 - effect of an error argument 316
 - error message 479
- FDXPI# 310
 - error message 478
- file
 - allocation, dynamic
 - error messages 497
 - with FILEINF routine 327
 - with INQUIRE statement 148
 - with OPEN statement 169
 - characteristics 327
 - connection 49, 50, 147
 - determining 150
 - determining for a named file 147
 - determining for an unnamed file 150
 - deallocation
 - error messages 497
 - deletion error message 447
 - existence 49
 - determining for a named file 147
 - determining for an unnamed file 150
 - information 327
 - name, unnamed 49
 - preconnection 50, 150
 - status 169
- FILEDEF command
 - error messages 497
- FILEINF routine 327
 - error messages 445
 - examples 331
 - rules 330
- FILEINF subroutine
 - error messages 473
 - with INQUIRE statement 148
 - with OPEN statement 169
- FIPS flagger 405
- fix 60
- fixed
 - length record 4
- fixed-form source
 - comments 11
 - continuation line 12
 - example 12
 - initial line 12
 - number 228
- fixed-view data-in-virtual subroutines 362
- fixed-view data-in-virtual subroutines, fixed-view 364
- FIXP8#
 - error message 489
- FIXPI# 310
 - error message 478
- flaggers, source language 405
- float 60
- floating-point 293
 - exceptions 293
 - information, saving 413
 - registers 413
- format codes
 - begin data transmission (T) 122
 - binary data transmission (B) 115
 - blanks
 - interpretation of (BN) 124
 - interpretation of (BZ) 124
 - character
 - constant transmission (H) 121
 - data transmission (A) 120
 - skipping (X) 122
 - comma 108
 - complex data editing (D,E,Q) 111
 - double precision data editing (D,E,Q) 111
 - end-of-record indication (/) 125
 - end-of-record inhibited indication (\$) 126
 - format
 - control termination (:) 126
 - specification reading 127
 - group format specification 123
 - hexadecimal data transmission (Z) 116
 - integer data editing (I) 110
 - list-directed formatting 128
 - logical variable transmission (L) 119
 - numeric 117
 - octal data transmission (O) 115
 - plus character control (S, SP, SS) 123
 - real data
 - editing (D,E,Q) 111
 - editing (F) 110
 - transmission (G) 112
 - rules for conversion 106
 - scale factor specification (P) 113
 - with comma as delimiter 108
 - with decimal point 108
- format control 106
- format identifier (FMT)
 - READ statement
 - direct access 186, 189
 - keyed access 190
 - sequential access 194
 - WRITE statement
 - direct access 239
 - keyed access 241
 - sequential access 244
- FORMAT statement 104
 - : code 126
 - / code 125
 - \$ code 126
 - A code 120
 - B code 115

FORMAT statement (*continued*)

- BN code 124
- BZ code 124
- colon code 126
- comma 108
- D code 111
- E code 111
- error messages 470
- F code 110
- forms of 109
- G code 112
- H code 121
- I code 110
- L code 119
- numeric code 117
- O code 115
- output 106
- P code 113
- printing 106
- Q code 111
- rules for conversion 106
- S code 123
- slash code 125
- SP code 123
- SS code 123
- T code 122
- X code 122
- Z code 116

FORTTRAN 66 410

FORTTRAN 77 410

FORTTRAN-supplied procedure

- See intrinsic functions

FQXP2# 310

- error message 482

FQXP8# 310

- error message 490

FQXPI# 310

- effect of an argument error 316
- error message 480

FQXPQ# 310

- effect of an argument error 316
- error messages 480

FQXPR error messages

- exponent out of range 482
- invalid argument value 487

free-form source

- comments 9
- continuation line 10
- continued line 10
- example 11
- initial line 10
- maximum length 10
- minus sign 10
- statement label 228

FRXP8# 310

- error message 490

FRXPI# 310

- error message 478

FRXPR# 310

- error message 479

function

- intrinsic 280

FUNCTION statement 131, 225

functions 50

- actual arguments 133
- dummy arguments 133
- END statement 92
- ENTRY statement 94
- implicitly called, exponential 314
- intrinsic
 - evaluating 32
- RETURN statement 217

functions, implicitly called

- complex multiply and divide 315

G

G format code 112

gamma and log gamma functions 300

- DGAMMA/DLGAMA 300
- GAMMA/ALGAMA 300

GAMMA/ALGAMA

- error message 487

general rules for data conversion 106

general service routines 318

generic functions 408

generic name of intrinsic function 280

global

- names 7
 - shortened form 8

GO TO statement 135

- assigned 136
- computed 137
- unconditional 137

group format

- nesting 107
- specification 123

H

H format code 121

hexadecimal

- constant 21
- data transmission 116

HFIX intrinsic function 286

hierarchy of operations

- arithmetic 34
- logical 43

Hollerith constants 20, 408

hyperbolic sine and cosine functions 301

- DSINH/DCOSH 301
- QSINH/QCOSH 301

hyperbolic sine and cosine functions (*continued*)

SINH/COSH 301

hyperbolic tangent functions 301

DTANH 301

QTANH 302

TANH 301

I

I format code 110

I/O

error messages 465

asynchronous I/O not supported 463

direct 476

direct access mode 463

end of record 471

fixup routine 492

parallel subroutine 495

record format invalid 471

referring to an unopened unit 455

referring to non-VSAM KSDS file 455

rewinding the file 487

sequential 476

statement processing 455

subtask abended 470

unopened VSAM file 464

VSAM file 464

WAIT statement 487

I/O statement 48

BACKSPACE 63

CLOSE 71

ENDFILE 92

FORMAT 104

I/O semantics 49

implied DO 90

INQUIRE 147

OPEN 168

PRINT 180

READ 183

REWIND 219

underlying semantics 49

WAIT 234

WRITE 236

I/O unit

connected to external file 175

information, output format 426

not connected to external file 175

PRINT statement 90

READ statement 90

status information, output format 427

WRITE statement 90

I*4

extend 60

IABS 295

IAND intrinsic function 288

IBCLR intrinsic function 290

IBCOM error messages

I/O fixup routine 492

secondary entry into main routine 492

IBCO error message 470

IBITS intrinsic function 290

IBM

extensions, documentation of xv

IBSET intrinsic function 290

ICHAR intrinsic function 288

identifying

data-in-virtual 364, 368

statements 228

user-supplied subprogram 103

VSAM linear data set 364

IEOR intrinsic function 288

IF statement 138

arithmetic 138

block 138

level 138

logical 142

IF-block 139

IGNFHDD routine 334

IGNFHU routine 334

IMAG intrinsic function 285

IMPLICIT statement 143

with double-byte characters 144

implicitly called functions

complex multiply and divide 315

implicitly called routines 310

assembler information 416

exponential 314

implied DO

error message 470

in DATA statement 89

in PRINT statement 90

in READ statement 90

in WRITE statement 90

INCLUDE directive 51, 145

INCLUDE statement

rules 146

INDEX intrinsic function 288

industry standards xv

initial line

fixed-form 12

free-form 10

initializing the execution environment 413

inline comment 11

input

format

fixed-form 4

free-form 4

records 4

INQUIRE statement

by file 147

by unit 150

INQUIRE statement (*continued*)
 by unnamed file 150
 default ddnames 151
 error statement, FILE specifier 466
 examples 162
 optional specifiers 152, 160
 with double-byte characters 153
 with dynamic allocation 148
 INT intrinsic function 286
 integer
 constant 14
 data editing 110
 data type 408
 value error message 470
 variable 186
 READ statement, direct access, formatted 186
 storage length 24
 INTEGER type statement 99
 integer-to-integer 310
 internal file
 error message 469
 International Organization for Standardization
 (ISO) 405
 intrinsic functions 408
 definition 3
 explicitly called 310
 reference 225
 INTRINSIC statement 163
 invalid VS FORTRAN Version 2 programs 2
 IOR intrinsic function 288
 IOSTAT
 parameters honored for I/O errors 392
 ISHFT intrinsic function 289
 ISHFTC intrinsic function 289
 ISO standard 405
 iteration count 88
 iteration variable 269

K

KEEP 72
 keyed access
 input/output 408
 INQUIRE statement 155
 READ statement
 formatted 189
 unformatted 211
 WRITE statement
 formatted 241
 unformatted 260
 keywords 13
 KSDS error message 463

L

L format code 119
 labeled
 DO 86
 DO WHILE 86
 LANGLVL(66) features 410
 LANGLVL(77) features 410
 language 405
 elements 405
 flaggers 405
 syntax 4
 LASCN error message 483
 LATN2 error message 482
 LEN intrinsic function 288
 length
 of statement 10, 12
 specification, IMPLICIT statement 143
 lexical compare error message 468
 LEXP error message 482
 LGAMA error message 489
 LGE intrinsic function 288
 LGT intrinsic function 288
 libraries
 availability 412
 error procedures 438
 interrupt procedures 438
 messages 438
 library
 messages 438
 reference 309
 linear data set, identifying 364
 list-directed formatting 128
 list-directed I/O 408
 PRINT statement 180
 READ statement
 external 200
 internal 202
 WRITE statement
 external 249
 internal 252
 literal constants 408
 LLE intrinsic function 288
 LLOG error message 482
 LLT intrinsic function 288
 load module
 processing termination 326
 LOC intrinsic function 288
 local
 names 7
 LOCAL (loop) statement 270
 LOCAL (section) statement 275
 lock
 function (PLCOND) 353
 request contention 351
 routines
 PLFREE routine 354

- lock (*continued*)
 - routines (*continued*)
 - PLLOCK routine 352
 - PLORIG routine 351
 - PLTERM routine 355
- LOG intrinsic function 281
- LOG/LOG10 302
- LOG10 intrinsic function 281
- logarithmic functions 302
 - ALOG/ALOG10 302
 - CLOG/CDLOG 302
 - CQLOG 303
 - DLOG/DLOG10 302
 - LOG/LOG10 302
 - QLOG/QLOG10 302
- logical
 - assignment statement 61
 - constant 18
 - data type 408
 - expression 41, 43
 - order of computations 43
 - IF statement 142
 - operator 41
 - variable
 - storage length 24
 - transmission 119
- logical primary
 - See primary
- LOGICAL type statement 99
- LSCN error message 482
- LSHIFT intrinsic function 289
- LSQRT error message 482
- LTNCT error message 483
- LXCMP error message 468

M

- main program
 - assign a name to 182
 - definition 2
 - PROGRAM statement 182
 - statement (PROGRAM) 50
- major language elements 405
- mathematical
 - exception tests 319
 - library routines 310, 412, 417
 - implicitly called 310
 - result registers 417
 - use in assembler language 412
 - use in VS FORTRAN Version 2 310
 - library, alternative 317
 - routine results 417
 - subroutine errors 397
- MAX intrinsic function 287
- message fragment 426

- messages
 - execution error 441
 - library 438
 - operator 439
 - program interrupt 438
- MIN intrinsic function 287
- minus sign 6, 10
- mixed character constant
 - definition 19
- mixed-mode arithmetic expressions 408
- MOD intrinsic function 285
- multiply complex numbers 310
- multitasking facility (MTF) 401
 - error unit 149, 151, 171
 - extended common and 401
 - print unit 149, 151, 171
 - sharing a common block 403
- MVBITS routine 343
- MVS considerations
 - operating system support xvi

N

- name
 - generic function 280
 - specific function 280
- named files 49
- NAMELIST statement 164
 - error messages 474
 - examples 166
 - input data 165
 - output data 167
 - READ statement
 - external 204
 - internal 206
 - WRITE statement
 - external 254
 - internal 256
- names
 - array 73, 85
 - block data 8, 65
 - CALL statement 94
 - common-block 8
 - constant 107
 - DBCS 9
 - ddname 49
 - definition 8
 - EBCDIC 9
 - elements of a program 7
 - example 9
 - external procedure 8
 - file 49
 - function 8
 - function reference 94
 - generic 163
 - global 7

- names (*continued*)
 - invalid 9
 - local 7
 - main program 8
 - specific 163
 - subroutine 8
 - valid 9
 - variable 23, 73
- naming conventions, default ddnames 151
- new page 106
- NINT intrinsic function 284
- non-mathematical arguments 319
- nonexecutable statement 3
- nonrecoverable failure, sample program and output 436
- NOT intrinsic function 288
- NPROCS (virtual processor query) 356
- NTASKS subroutine 401
- NULLIFY statement 167
- numeric
 - format code examples 117
 - input field 128

O

- O format code 115
- object-time dimensions 29
- obtaining a dynamic common length 371
- octal
 - constant 20
 - data transmission 115
- OPEN statement 168
 - connected units 175
 - double-byte characters 172
 - dynamic allocation 169
 - error messages
 - ACCESS specifier 466
 - BLANK specifier 466
 - conflicting specifiers 453
 - duplicate file name 455
 - empty file 456
 - FILE specifier 466
 - FORM specifier 466
 - invalid ACTION specifier 457
 - issued for file already open 452
 - KEYS specifier 454, 457
 - no available file 456
 - status of scratch 467
 - STATUS specifier 466
 - examples 175
 - for loading records 172
 - for reading records 173
 - for retrieving records 173
 - for writing records 172
 - nonconnected units 175

- operating system
 - support xvi
- operator
 - messages 439
- operators, arithmetic 33
- option default 392
- option table
 - default values 392
 - entry 389
 - error 382
- order of statements in a program unit 51
- ORIGINATE statement 264
- output
 - format 423
 - array 425
 - control flow information 426
 - I/O unit status information 427
 - variable character 424
 - variable noncharacter 423
 - from symbolic dumps 423
- OVERFL routine 319
- overflow
 - error message 440
 - floating-point 293
 - premature 293
 - terminal 293
 - indicator service routine 319

P

- P format code 113
- packets, debug 81
- page control 106
- parallel
 - call statements
 - CALL statement 67
 - PARALLEL CALL statement 277
 - WAIT FOR ALL CALLS statement 278
 - considerations
 - common blocks 74
 - data-in-virtual subroutine calls 362
 - DO variable 87
 - error option table 382
 - INQUIRE by unnamed file 151
 - MTF subroutine 362
 - RETURN statement 217
 - SAVE statement 224
 - XUFLOW subroutine 320
 - event cycle operations 345
 - event identifier 346
 - event routines
 - PEORIG routine 346
 - PEPOST routine 348
 - PETERM routine 348
 - PEWAIT routine 348
 - language constructs 264

- parallel (*continued*)
 - lock
 - contention 351
 - function (PLCOND) 350
 - function value (fv) 351
 - request contention 351
 - lock routines
 - PLFREE routine 354
 - PLLOCK routine 352
 - PLORIG routine 351
 - PLTERM routine 355
 - loop statements
 - DOAFTER statement 272
 - DOBEFORE statement 271
 - DOEVERY 272
 - EXIT statement 273
 - LOCAL (loop) statement 270
 - PARALLEL DO statement 269
 - processing 344
 - root task 265
 - sections statements
 - END SECTIONS statement 276
 - LOCAL (section) statement 275
 - PARALLEL SECTIONS statement 274
 - SECTION statement 275
 - service routines
 - CPUTIME, warning 338
 - statements 47
 - subroutines
 - DSPTCH subroutine 402
 - NTASKS subroutine 401
 - SHRCOM subroutine 403
 - SYNCRO routine 403
 - task 264
 - task management statements
 - ORIGINATE statement 264
 - SCHEDULE statement 265
 - TERMINATE statement 268
 - WAIT FOR statements 267
 - thread 278, 344
 - virtual processor query, NPROCS 344, 356
- PARALLEL CALL statement 277
- PARALLEL DO statement 269
 - LOCAL iteration variable 87
 - passed by copy 68
- PARALLEL SECTIONS statement 274
- PARAMETER statement 176
- parameters, correct 414
- parentheses
 - error message 462
 - in arithmetic expressions 35
 - types 47
 - use in character expressions 39
 - use in logical expressions 45
- parenthesized primary 266, 277
- passed by copy 68, 267, 278
- PAUSE statement 178
 - error message 439
- PEORIG event routine 346
- PEPOST event routine 348
- period xiii, 6
- PETERM event routine 348
- PEWAIT event routine 348
- PLCOND lock function 350, 353
- PLFREE lock routine 354
- PLLOCK lock routine 350, 352
- PLORIG lock routine 350, 351
- PLTERM lock routine 355
- plus sign 6
- pointee array 28
- pointee variable
 - allocating space for 54
 - defining 178
 - in a parallel loop 270
 - nullifying 167
 - releasing space for 78
- POINTER statement 178
- pointer variable
 - defining 178
 - nullifying 167
 - resetting to unassigned state 78, 167
 - restrictions 179
 - use 179
- positive
 - assign 60
- preconnected file
 - definition 3
 - OPEN statement 50
 - WRITE statement
 - formatted with direct access 240
 - formatted with sequential access 245
 - list-directed with external devices 251
 - NAMELIST with external devices 255
 - unformatted with direct access 259
 - unformatted with sequential access 263
- preserving computational independence 267
- preserving shift codes 340
- primary 33
 - logical 41
- PRINT statement 180
 - implied DO in 90
- printouts, sample storage 423
- procedure
 - definition 3
 - external 3
- PROCESS statement 52
- processing termination 326
- program
 - basic elements of 2
 - executable 3
 - interrupt error 441

- program (*continued*)
 - main 2
 - mask 419
 - unit
 - definition 2
 - order of statements in 51
- PROGRAM statement 50, 182
- program-interrupt messages 396, 438
- publications
 - summary of xiii

Q

- Q format code 111
- QARCOS 297
- QARSIN/QARCOS, error message 488
- QASCN error message 488
- QATAN/QATAN2
 - error message 488
- QERFC 299
- QEXP
 - error message 487
- QEXT intrinsic function 286
- QLOG/QLOG10
 - error message 488
- QP
 - extend 60
 - float 60
- QSCN error message 488
- QSCNH error message 488
- QSIN/QCOS, error message 488
- QSINH/QCOSH error message 488
- QSQRT
 - error message 487
- QTNCT error message 488, 489
- quotation mark 6, 9
 - with character constant 19

R

- range
 - of a DO loop 87
 - of a DO WHILE loop 87
- range of the DO 86
- READ statement
 - asynchronous 183
 - error messages 454
 - file not created 476
 - implied DO 470
 - KEY argument 454
 - formatted with
 - direct access 186
 - keyed access 189
 - sequential access 193
 - sequential access, internal 196
 - forms of 183

- READ statement (*continued*)
 - implied DO in 90
 - NAMELIST with
 - external devices 204
 - internal files 206
 - unformatted with
 - direct access 209
 - keyed access 211
 - sequential access 215
- real
 - assign 60
 - constant 15, 16
 - examples 16
 - data of length 8 24
 - data type 409
 - exponent 16
 - variable, storage length 24
- REAL intrinsic function 286
- REAL type statement 99
- real-to-integer 310
- real-to-real 310
- record
 - error messages
 - record length 463
 - record number 476
 - format
 - fixed length 4
 - variable length 4
- related publications xiii
- relational
 - expression 39, 407
 - operators 39, 409
- repositioning files 63
- request traceback 388
- resetting in data-in-virtual 373
- return code routines
 - SYSRCS 324
 - SYSRCT 324
 - SYSRCX 324
- RETURN statement
 - in a function 217
 - in a subroutine 218
- REWIND statement 219
- REWRITE statement
 - formatted 220
 - unformatted 222
- rewriting records 220
- root task 265
- routines
 - explicitly called 310
 - implicitly called
 - character 313
 - mathematical 310
 - service 314
 - service and utility routines 319
 - subroutine
 - error messages 462

- RSHIFT intrinsic function 289
- rules
 - arithmetic expression construction 33
 - data conversion 106
 - language 2
- run-time
 - environment, initialization 413
 - termination 326

S

- S format code 123
- SAA flagger 405
- sample
 - programs and output 428
 - array items 431
 - variable items 428
 - storage printout 423
- SASCN error message 481
- SATN2 error message 481
- save option table entry 384
- SAVE statement 224
 - SAVE statement 224
- saving changes in data-in-virtual 372
- scale factor 113
- SCHEDULE statement 265
- SCHEDULE statement identifiers
 - arg 266
 - cpcom 266
 - cpicom 266
 - cpocom 266
 - ptaskid 266
 - shrcom 266
 - subx 266
- SCNH error message 483
- SDUMP routine
 - description 323
 - format specifications 323
 - message fragment 426
 - output 323
 - output format 423
 - output of symbolic dump 427
 - programming considerations 323
 - sample printouts 423
- SECTION statement 275
- sequential access
 - INQUIRE statement 159
 - READ statement
 - formatted 193
 - unformatted 215
 - WRITE statement
 - formatted 243
 - unformatted 262
- sequential I/O 409
- serializations
 - compile-time, parallel 274
- service routines 319
 - allocation, dynamic
 - FILEINF 327
 - ARGSTR 342
 - assembler information 417
 - ASSIGNM 340
 - CDUMP/CPDUMP 322
 - CLOCK 336
 - CPUTIME 337
 - data-in-virtual subroutines 361
 - DIVCML 371
 - DIVINF 364
 - DIVINV 368
 - DIVRES 373
 - DIVSAV 372
 - DIVTRF 367
 - DIVTRV 372
 - DIVVWF 366
 - DIVVWV 369
 - DATIM 339
 - DUMP/PDUMP 321
 - end processing 326
 - error-handling subroutines 382
 - ERRMON subroutine 383
 - ERRSAV subroutine 384
 - ERRSET subroutine 385
 - ERRSTR subroutine 388
 - ERRTRA subroutine 388
 - EXIT 326
 - IGNFHDD 334
 - IGNFHU 334
 - mathematical exception test 319
 - multitasking facility (MTF) subroutines 401
 - DSPTCH 402
 - NTASKS 401
 - SHRCOM 403
 - SYNCRO 403
 - MVBITS 343
 - return code routines 324
 - SYSRCS 324
 - SYSRCT 324
 - SYSRCX 324
 - root task 326
 - SDUMP 323
 - storage dump 321
 - SYSABD 326
 - SYSABN 326
 - UNTANY 331
 - UNTNOFD 331
 - XUFLOW 320
- seventy-seven language level 405
- SEXP error message 480
- SGAMA error message 487
- shareable load 459, 460
- sharing
 - a common block 403

- shift-out/shift-in characters
 - documentation xiii
 - I/O support 172
 - in continued source lines 10, 12
 - in source 12
- SHRCOM subroutine 403
- SIGN intrinsic function 285
- SIN
 - error message 481
- SIN intrinsic function 282
- SIN/COS error message 481
- sine and cosine functions 303
 - CDSIN/CDCOS 304
 - COS 303
 - CQSIN/CQCOS 305
 - CSIN/CCOS 304
 - DCOS 304
 - DSIN 303
 - QSIN/QCOS 304
 - SIN 303
- SINH intrinsic function 283
- SINH/COSH, error message 481
- sixty-six language level 405
- size and type declaration of an array 27
- skip a line 106
- slash 6
- slash format code 125
- SLOG error message 480
- source language flaggers 405
- source language statement
 - fixed-form 11
 - free-form 9
- source statement characters
 - digit 6
 - letter 6
 - special characters 6
- SP format code 123
- space 413
- special characters 6
- specific name of intrinsic function 280
- specification statement 50
 - AUTOMATIC 63
 - BYTE type statement 99
 - CHARACTER type 99
 - COMMON 73
 - COMPLEX type 99
 - DIMENSION 85
 - DOUBLE COMPLEX type 99
 - DOUBLE PRECISION type 99
 - EQUIVALENCE 97
 - explicit type 99
 - EXTERNAL 103
 - IMPLICIT 143
 - INTEGER type 99
 - INTRINSIC 163
 - LOGICAL type 99
- specification statement (*continued*)
 - NAMELIST 164
 - PARAMETER 176
 - REAL type 99
 - SAVE 224
 - STATIC 228
 - UNSIGNED type statement 99
- SQRT
 - error message 480
- SQRT intrinsic function 285
- square root functions 305
 - CQSQRT 306
 - CSQRT/CDSQRT 305
 - DSQRT 305
 - QSQRT 305
 - SQRT 305
- square root routines
 - example with vector registers 419
 - examples 413
- SS format code 123
- SSCN error message 481
- SSCNH error message 481
- SSQRT error message 480
- standards for language 405
- standards, industry xv
- statement
 - action 3
 - categories 47
 - descriptions 52
 - executable 3
 - explicit 24
 - fixed-form number 11, 228
 - free-form label 228
 - function 409
 - function statement 225
 - functions of 47
 - implicit 24
 - inserting with INCLUDE 145
 - label 12, 228
 - READ statement, direct access, formatted 186
 - length 10, 12
 - maximum length 10
 - nonexecutable 3
 - number 55, 228
 - order 51
 - specification 24
- static
 - in equivalence 97
- STATIC statement 228
- status of a file 169
- STNCT error message 481
- stop display 233
- stop program 91
- STOP statement 229
 - error message 439

- storage
 - See *also* utility routines
 - dump service routines 321
 - estimates, extended precision routines 413
 - printout, sample 423
 - shared 73, 97
- storage dump 321
- store entry in option table 388
- SUBCHK function of DEBUG 81
- subprogram
 - FUNCTION statement 131
- subprograms 50
 - BLOCK DATA statement 51
 - calling in assembler 413
 - definition 2
 - FUNCTION statement 50
 - initialization instructions 413
 - statement 51
 - SUBROUTINE statement 50, 229
- SUBROUTINE statement 229
- subroutine subprogram
 - actual arguments 230
 - dummy arguments 231
 - END statement 92
 - ENTRY statement 94
- subroutines
 - actual arguments 230
 - dummy arguments 231
 - END statement 92
 - ENTRY statement 94
 - general service 318
 - RETURN statement 218
- subscripts
 - definition 26
 - examples 27
 - identify array elements 26
 - in DATA statement 76
- substring
 - character 30
 - expression 30
 - in DATA statement 76
- summary of publications xiii
- symbolic dumps
 - array items 431
 - examples 427
 - how to call 323
 - nonrecoverable failure 436
 - output format 423
 - variable items 427
- symbolic names
 - See names
- synchronization of parallel event 345
- SYNCRO routine 403
- syntax 4
 - notation xii

- SYSABD routine 326
- SYSABN routine 326
- SYSRCS routine 324
- SYSRCT routine 324
- SYSRCX routine 324
- system
 - support, operating xvi
- Systems Application Architecture flagger 405
 - See *also* SAA

T

- T format code 122
- tables, error option 382
- TAN intrinsic function 282
- TAN/COTAN error message 481
- tangent and cotangent functions 306
 - COTAN 306
 - DCOTAN 307
 - DTAN 306
 - QTAN/QCOTAN 307
 - TAN 306
- TANH intrinsic function 283
- task
 - routines 318
- terminal statement of a DO loop 86
- terminate processing (EXIT) 326
- TERMINATE statement 268
- terminating
 - access in data-in-virtual 367, 372
- test
 - exponents 319
 - for divide-check exception 319
- time routines
 - CLOCK and CLOCKX 336, 339
 - DATIM 339
 - DATIMX 339
- TRACE OFF statement 233
 - in debug packet 81
- TRACE ON statement 234
 - in debug packet 81
- traceback 436
 - request 388
- trigonometric
 - subprograms, error message 481
- TRUE 18
- two (2) raised to real power 310
- type declaration
 - by predefined specifications 25
 - DBCS names 25
 - declaration of an array 27
 - EXPLICIT statement 26
 - IMPLICIT statement 25
 - predefined 25
- type specification 143

typeless constant
 definition 14
typless constant
 using 22

U

unary signs 34
unconditional GO TO statement 137
underflow
 error message 440
 floating-point 293
 premature 293
 terminal 293
underscore 6
unformatted I/O error message 471
unit
 connected 150
 connected to external file 175
 connection, determining 150
 INQUIRE statement 150
 not connected to external file 175
 number 150, 168
 OPEN statement 168
UNIT specifier
 error messages 465
 on the INQUIRE statement 150, 151
unlabeled
 DO 86
 DO WHILE 86
unnamed files 49
unsigned
 data type 409
unsigned integer
 variable
 storage length 24
UNSIGNED type statement 99
UNTANY routine 331
UNTNOFD routine 331
updating records 220
user exit routine, coding the 386
utility and service subroutines
 CDUMP/CPDUMP 322
 DUMP/PDUMP 321
 EXIT 326
 SDUMP 323
 XUFLOW 320

V

valid VS FORTRAN Version 2 programs 2
value separator 128
VAMTP error message 496
variable character, output format 424
variable items, sample program and output 428

variable noncharacter, output format 423
variable-length records 4
variables
 character 30
 declaration 24
 definition 23
 lengths 23
 names 23
 types 23
varying-view data-in-virtual subroutines 362
VASYN error messages, unit number out of range 474
VASYP error messages
 asynchronous I/O
 ddname 468
 not supported 463
 subtask 470
 blocksize not specified 477
 end of data set 472
 end of record 471
 I/O error 472
 invalid address 475
 no wait issued for outstanding I/O request 487
 REWIND statement 487
 unformatted I/O 471
 WAIT statement 487
VCIA4 error message 469
VCLOP error message 467
VCOM2 error messages
 I/O fixup routine 492
VCOMH error messages
 end of record 471
 invalid character in FORMAT statement 470
 nested parentheses overflow 462
VCTVH error messages
 extended integer value out of range 491
 illegal binary character 475
 illegal decimal character 472
 illegal hexadecimal character 475
 illegal octal character 475
 integer value out of range 470
 real value out of range 475
VDIOS error messages
 direct access
 data set 476
 I/O 476
 read 476
 direct/keyed access I/O statement 495
 FORMATTED and UNFORMATTED I/O
 requests 465
 I/O error 472
 incorrect record length specified 477
 OPEN request ignored 465
 OPEN statement ACTION specifier conflict 453
 record
 length 476
 number 476
 number and length 460

VDIOS error messages (*continued*)
 SCRATCH file 465
 unit number out of range 474
 VDIVP error messages 458
 vector
 count register 418
 entry points 420
 error handling 419
 interruption index 419
 intrinsic function
 calling sequence 417
 result registers 418
 mask register 418
 registers, passing arguments in 417
 VEMGN error message 492
 error count 491
 error monitor 491
 text for error message 492
 VFNTH error messages
 divide by zero 441
 overflow 440
 program interruption 441
 underflow 440
 VIOS error message 469
 VINQP error message 466, 473
 VINTP error messages
 address conflict in link mode 458
 composite module conflict 458
 module loaded not recognized 459
 secondary entry into main routine 492
 specifier list conflict 461
 VIOCP error message 492
 VIOFP error message 492
 VIOLP error messages
 end of record 471
 error in repeat count 475
 incorrect delimiter 477
 item size exceeds buffer length 470
 VIONP error messages
 end of record 474
 invalid subscript 475
 NAMELIST statement 474
 VIOUF/VIOFM error message 455
 VIOUP error messages 469
 end of record 471
 I/O fixup routine 492
 virtual processor query (NPROCS) 356
 VKIOS error messages
 add record with duplicate key 456
 direct/keyed access I/O statement 495
 file name 455
 I/O
 statement processing 455
 statement refers to non-VSAM KSDS file 455
 inconsistent record lengths 456
 key
 argument length too long 454

VKIOS error messages (*continued*)
 key (*continued*)
 of reference conflict 457
 specifies no available file 456
 KEYID specifier conflict 454
 open an empty file 457
 OPEN statement
 ACTION specifier conflict 453
 with conflicting specifiers 453
 record
 loading 458
 not found with specified key 454
 too short 455
 VLCIO error messages
 abend 492
 library initialization 497
 VLINP error messages
 load module does not contain shareable part 460
 load module has incorrect format 460
 sort timestamp conflict 460
 unable to branch to NONSHAREABLE part 459
 VMEXP error messages
 addressing mode conflict 496
 multitasking facility 497
 multitasking facility function 494
 parallel subroutine load module 495
 VMIN error messages
 main task/load module conflict 496
 VMINI error messages
 parallel subroutine load module 494
 VMMAA error messages
 autotask dd statement 494
 multitasking facility subtask 495
 parallel subroutine load module 494
 VMNTP error messages
 multitasking facility function invalid 494
 multitasking facility function without arguments 494
 VMOPP error messages
 error number out of range 467
 unmodifiable message table entry 469
 VMPRM error message 493
 VMSYP error message 494
 VMTRM error messages
 multitasking facility 497
 multitasking facility subtask 495
 parallel subroutine 496
 VOCMP error messages 462
 VOCMP#
 error message 461
 VOPEP error messages 473
 ACTION specifier 457
 invalid specifier 466
 access 466
 blank 466
 file 466
 form 466
 status 466

VOPEP error messages (*continued*)

- KEYS specifier conflict 457
- OPEN statement issued for file already open 452
- recl specifier conflict 461
- status of scratch not allowed 467

VPARM error messages

- autotask keyword 475
- autotask keyword subparameter 493
- debunit keyword subparameter 493
- invalid autotask keyword 493

VPIOS error messages

- I/O 472

VPIOS message

- BACKSPACE statement 445

VSAM

- error messages 463
- linear data set, identifying 364

VSCOM error messages

- I/O fixup routine 492
- secondary entry into main routine 492

VSERH error message 475

VSIOS error messages 453

- CLOSE statement 464
- direct I/O 476
- end of data set 472
- ENDFILE statement 463
- FORMATTED and UNFORMATTED I/O requests 465
- I/O error 472
- I/O in a parallel subroutine 495
- need REWIND statement 453
- OPEN request ignored 465
- OPEN statement ACTION specifier conflict 453
- REWIND statement 487
- SCRATCH file 465
- sequential I/O 476
- unformatted I/O 471
- unit number out of range 474

VSIOS message

- BACKSPACE statement 445

VSIOS/VDIOS error message 460

VSPAP error message 467

VSPIP error message 497

VSTAE error message 477

VSTIO error messages 497

VVIOS error messages 472

- ENDFILE statement 463
- file positioning I/O statement in direct access mode 463
- I/O request made to unopened VSAM file 464
- I/O statement 464
- key sequenced data set 463
- OPEN statement ACTION specifier conflict 453
- record addition conflict 463
- record length too long 463
- record number 476

VVIOS error messages (*continued*)

- unit number out of range 474

W

WAIT FOR ALL CALLS statement 278

WAIT FOR ALL TASKS statement 267

WAIT FOR ANY TASK statement 267

WAIT FOR statements

- WAIT FOR ALL TASKS statement 267
- WAIT FOR ANY TASK statement 267
- WAIT FOR TASK statement 267

WAIT FOR TASK statement 267

WAIT statement 234

- error messages 487
- valid 236

write an end-of-file record 92

WRITE statement 236

- asynchronous 237
- error messages 463, 470
- formatted with
 - direct access 238
 - keyed access 241
 - sequential access 243
 - sequential access, internal 246
- forms of 236
- implied DO in 90
- list-directed I/O with
 - external devices 249
 - internal files 252
- NAMelist with
 - external devices 254
 - internal files 256
- unformatted with
 - direct access 257
 - keyed access 260
 - sequential access 262

X

X format code 122

XOR intrinsic function 288

XUFLOW

- routine 320

Z

Z format code 116

We'd Like to Hear from You

VS FORTRAN Version 2
Language and Library Reference
Release 6
Publication No. SC26-4221-08

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use one of the following network IDs:
 - IBMLink: HLASMPUB at STLVM27
 - Internet: COMMENTS@VNET.IBM.COM

Be sure to include the following with your comments:

- Title and publication number of this book
- Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

VS FORTRAN Version 2
Language and Library Reference
Release 6

Publication No. SC26-4221-08

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Grammatically correct and consistent	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Graphically well designed	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

Company or Organization

Phone No.



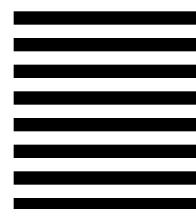
Fold and Tape

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, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Department J58
International Business Machines Corporation
PO BOX 49023
SAN JOSE CA 95161-9945



Fold and Tape

Please do not staple

Fold and Tape



File Number: S370-40
Program Number: 5668-805
5668-806
5688-087

Printed in the U.S.A.

The VS FORTRAN Version 2 Library

LY27-9516 Diagnosis Guide
GC26-4219 General Information
SC26-4340 Installation and Customization for MVS
SC26-4339 Installation and Customization for CMS
SC26-4420 Installation and Customization for AIX/370
SC26-4223 Interactive Debug Guide and Reference
SC26-4221 Language and Library Reference
GC26-4225 Licensed Program Specifications
SC26-4603 Master Index and Glossary
SC26-4686 Migration from the Parallel FORTRAN PRPQ
SC26-4741 Programming Guide for AIX/370
SC26-4222 Programming Guide for CMS and MVS
SX26-3751 Reference Summary

SC26-4221-08





VS FORTRAN Version 2

Language and Library Reference

Release 6