VS FORTRAN Language and Library Reference

Program Product

```
IRR(I,J) = 1
IRI(I,J) = 2
CONTINUE
PRINT 20, (
1 I = 1, 3)
FORMAT (3(1X
STOP
END
```

Program Numbers:

5748-FO3 Compiler and Library
5748-LM3 Library Only

Release 4.0

IBM

VS FORTRAN Language and Library Reference

Program Numbers
5748-FO3 (Compiler and Library)
5748-LM3 (Library Only)
Release 4.0

SC26-4119-0

First Edition (October 1984)

This edition applies to Release 4.0 of VS FORTRAN, Program Products 5748-FO3 (Compiler and Library) and 5748-LM3 (Library only), and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

New features for this release are summarized under "Summary of Amendments" following the preface. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

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

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

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California, U.S.A. 95150. 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 1981, 1982, 1983, 1984

Preface

The VS FORTRAN Compiler and Library, Version 1, Release 4.0, program product is commonly referred to as Level 1.4.0. It is known as Release 4.0 in this manual.

This manual outlines the programming rules for VS FORTRAN 1978-level source language. It includes Full American National Standard FORTRAN (X3.9-1978), plus IBM extensions.

After an introduction, Part 1, "Language Reference," discusses:

- VS FORTRAN Language
- VS FORTRAN Data
- VS FORTRAN Expressions
- VS FORTRAN Statements (in alphabetic order)
- VS FORTRAN Intrinsic Functions

After an introduction, Part 2, "Library Reference," discusses:

- Mathematical, Character, and Bit Subprograms
- Service Subroutine Subprograms
- Extended Error Handling Subroutines and Error Option Table

The appendixes contain the following additional information:

A. Source Language (FIPS) Flagger (including execution-time cautions)
B. IBM and ANS FORTRAN Features
C. EBCDIC and ASCII Codes
D. Algorithms for Library Mathematical Functions
E. Storage Estimates
F. Accuracy Statistics
G. Assembler Language Information
H. Sample Storage Printouts
I. Library Procedures and Messages
J. Module Names

Industry Standards

The VS FORTRAN Compiler and Library program product is designed according to the specifications of the following industry standards, as understood and interpreted by IBM as of May, 1982.

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

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

The bit string manipulation functions are defined in ANSI/ISA-S61.1.

The following two standards are technically equivalent. In this manual, 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

Both the FORTRAN 77 and the FORTRAN 66 standard languages include IBM extensions. In this book, 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.

Related Publications

VS FORTRAN publications are designed to help develop programs with a minimum of wasted effort. This book, *VS FORTRAN Language and Library Reference*, describes the rules for coding VS FORTRAN programs when using the current FORTRAN. It also contains detailed information about the execution-time library subroutines.

VS FORTRAN Publications

Other VS FORTRAN publications contain related information.

- *VS FORTRAN Compiler, Library, and Interactive Debug General Information*, GC26-4114, contains information that is intended as an aid to evaluating and planning for the use of the VS FORTRAN Compiler and Library program products.
- *VS FORTRAN Compiler and Library Installation and Customization*, SC26-3987, contains material for installing the VS FORTRAN Compiler and

Library and is to be used in conjunction with the VS FORTRAN Program Directory that applies to your system.

- *VS FORTRAN Programming Guide*, SC26-4118, contains guidance information on designing, coding, debugging, testing, and executing VS FORTRAN programs written at the current FORTRAN language level. In addition, separate chapters discuss executing your FORTRAN program under VM/SP, under MVS/SP, including MVS/XA, under VSE/Advanced Functions, and under VM/PC.

- *VS FORTRAN Compiler and Library Reference Summary*, SX26-3731, is a pocket-sized reference booklet containing current FORTRAN syntax and brief descriptions of the compiler options.

- *VS FORTRAN Compiler and Library Diagnosis*, SC26-3990, tells you how to diagnose failures in the VS FORTRAN Compiler and Library.

In addition, a binder for VS FORTRAN publications and a combination of binder and publications are available.

- Binder only, SX26-3747

- Binder and the following publications, SBOF-1192

  - VS FORTRAN *Programming Guide*

  - VS FORTRAN *Language and Library Reference*

  - VS FORTRAN *Compiler and Library Reference Summary*

FORTRAN IV Publications

- *IBM System/360 and System/370 FORTRAN IV Language*, GC28-6515, describes the source language available in the FORTRAN IV language, and contains the rules for writing VS FORTRAN programs using FORTRAN 66.

- *FORTRAN Coding Form*, GX28-7327, aids in coding fixed-form FORTRAN programs.

VS FORTRAN Interactive Debug Publications

- *VS FORTRAN Compiler, Library, and Interactive Debug General Information*, GC26-4114 (see description above under "VS FORTRAN.")

- *VS FORTRAN Interactive Debug Guide and Reference*, SC26-4116

- *VS FORTRAN Interactive Debug Installation*, SC26-4117

- *VS FORTRAN Interactive Debug Reference Summary*, SX26-3742

- *VS FORTRAN Interactive Debug Diagnosis*, SY26-3944

System and Device Information

Specific system information and details about block size, track capacity, and so on, of the various input/output devices are **not** included in this book. See the following system publications for this information:

IBM DASD Publication

Introduction to IBM Direct Access Storage Devices and Organization Methods, GC20-1649, contains algorithms for direct files.

IBM-Supplied Utility Programs

OS/VS2 MVS Utilities, GC26-3902

MVS/Extended Architecture Utilities, GC26-4018

VSE/Advanced Functions System Utilities, SC33-6100

Assembler Language Programming

OS/VS–DOS/VSE–VM/370 Assembler Language, GC33-4010

OS/VS–VM/370 Assembler Programmer's Guide, GC33-4021

Guide to DOS/VSE Assembler, GC33-4024

Assembler H Version 2 Application Programming: Language Reference, GC26-4037

Assembler H Version 2 Application Programming: Guide, GC26-4036

System/370 Machine Characteristics

IBM System/370 Principles of Operation, GA22-7085. It describes the various types of interruptions.

OS/VS Systems Publications

OS/VS Linkage Editor and Loader, GC26-3813

OS/VS Virtual Storage Access Method (VSAM) Programmer's Guide, GC26-3838

OS/VS Tape Labels, GC26-3795

MVS Publications

OS/VS2 MVS Data Management Services Guide, GC26-3875

OS/VS2 Supervisor Services and Macro Instructions, GC28-0683

OS/VS2 Access Method Services, GC26-3841

OS/VS2 MVS JCL, GC28-0692

OS/VS2 Debugging Guide, GT28-0632

OS/VS2 TSO Terminal User's Guide, GC28-0645

OS/VS2 TSO Command Language Reference, GC28-0646

TSO-3270 Structured Programming Facility (SPF) Program Reference Manual, SH20-1730

MVS/Extended Architecture (MVS/XA) Publications

MVS/Extended Architecture Access Method Services Reference, GC26-4019

MVS/Extended Architecture Supervisor Services and Macro Instructions, GC28-1154

MVS/Extended Architecture JCL, GC28-1148

MVS/Extended Architecture Debugging Handbook, Vols. 1-5, GC28-1164-1168

MVS/Extended Architecture Data Management Services, GC26-4013

MVS/Extended Architecture Linkage Editor and Loader, GC26-4011

MVS/Extended Architecture VSAM Programmer's Guide, GC26-4015

MVS/Extended Architecture Tape Labels, GC26-4003

MVS/Extended Architecture TSO Command Language Reference, GC28-0646, as updated by Supplement SD23-0259

MVS/Extended Architecture TSO Extensions TSO Command Language Reference, SC28-1134

VM/CMS Systems Publications

VM/SP CP Command Reference for General Users, SC19-6211

VM/SP CMS User's Guide, SC19-6210

VM/SP CMS Command and Macro Reference, SC19-6209

VM/SP Terminal User's Guide, GC19-6206

VSE Publications

VSE/Advanced Functions System Management Guide, SC33-6094

VSE System Data Management Concepts, GC24-5209

VSE/Advanced Functions Tape Labels, SC24-5212

VSE/Advanced Functions DASD Labels, SC24-5213

VSE/Advanced Functions Macro User's Guide, SC24-5210

VSE/Advanced Functions Serviceability Aids and Debugging Procedures, GC33-6099

VSE/VSAM Programmer's Reference, SC24-5145

Using VSE/VSAM Commands and Macros, SC24-5144

Using the VSE/VSAM Space Management for SAM Feature, SC24-5192

Alternative Mathematical Library Subroutines

The Evaluation of Periodic Functions with Large Input Arguments, by Jesse Y. Wang, ACM/SIGNUM, December 1978

Argonne National Laboratory, Applied Mathematics Division, System/360 Library Subroutine:

ANL B357S-1	DEXP
ANL B457S-3	A**B (single-precision)
ANL B458S-1	A**B (double-precision)
ANL B356S-1	EXP
ANL B159S-3	DTAN/DCOTAN
ANL B158S-2	DSIN/DCOS

Methods of Presentation

Because methods of presentation vary from book to book, the format notation and method of indicating IBM extensions are outlined here.

Format Notation

In this manual, "must" is to be interpreted as a requirement; conversely, "must not" is to be interpreted as a prohibition.

In describing the form of VS FORTRAN statements or constructs, the following conventions and symbols are used:

- Special characters from the VS FORTRAN character set, uppercase letters, and uppercase words are to be written as shown, except where otherwise noted.

- Lowercase letters and lowercase words indicate general entities for which specific entities must be substituted in actual statements. After a given lowercase letter or word is used in a syntactic specification to represent an entity, all subsequent occurrences of that letter or word represent the same entity until that letter or word is used in a subsequent syntactic specification to represent a different entity.

- Square brackets ([ ]) are used to indicate optional items.
- An italicized word (or underlined in the examples) indicates a variable, such as an entry point, name of a function, data type, or list of variables or array names.
- An ellipsis (...) indicates that the preceding optional items may appear one or more times in succession.
- Blanks are used to improve readability; however, unless otherwise noted, they have no significance.
- For clarity of presentation, continuation designators have been omitted from continuation lines in example.

The general form of each statement is enclosed in a box. For example:

Syntax

CALL *name* [([*arg*1 [,*arg*2] ...])]

The following examples are among those allowed:

CALL *name*
CALL *name* ()
CALL *name* (*arg*)
CALL *name* (*arg*, *arg*)
CALL *name* (*arg*, *arg*, *arg*)
CALL *name* (*arg*, *arg*, *arg*, *arg*)

When an actual statement is written, specific entities are substituted for *name* and each *arg*. For example:

CALL ABCD (X,1.0)

Documentation of IBM Extensions

In addition to the statements available in FORTRAN 77, IBM provides "extensions" to the language. These extensions are shown in the following ways.

IBM Extension

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

End of IBM Extension

The following example shows how boxes indicate IBM extensions.

NAME	TYPE	LENGTH
I, J, K	Integer variables	4 [, 2, 2]
C	Real variable	4
D	Complex variable	[16]

The example below shows how IBM extensions are documented within a table. Boxes around certain types and lengths of the result of logical operations indicate IBM extensions.

Second Operand \ First Operand	[Logical (1)]	Logical (4)
[Logical (1)]	[Logical (4)]	[Logical (4)]
Logical (4)	[Logical (4)]	Logical (4)

Summary of Amendments

October 1984

Merger of VS FORTRAN Reference Manuals

VS FORTRAN Application Programming: Language Reference and *VS FORTRAN Application Programming: Library Reference* have been merged into this manual. The original manuals have become independent parts of this new one, and, with few exceptions, the kinds of information that were in them before can be found in the corresponding parts of this manual. However, common parts of the original manuals (tables of contents, indexes, and so on) have been consolidated.

Release 4.0 Enhancements

VSAM Key-Sequenced Data Sets

VS FORTRAN programs can now load and access VSAM key-sequenced data sets (KSDS):

- Records can be retrieved, added, replaced, and deleted, using key values (designated fields within the records).
- Both direct and sequential processing (by key value) are allowed.
- Multiple alternate keys, as well as a primary key, can be used.

The following language statements have been expanded:

- OPEN, so that you can open a file for keyed access. The length and location of the keys to be used are specified on this statement.
- READ, so that you can specify a key value for the retrieval of records. The key to be used in a multiple-key file is specified on this statement.
- WRITE, so that you can identify a statement as the one to be given control, if a duplicate key value is written.
- INQUIRE, so that you can find out the value of the last key used in an input/output operation, and which of multiple keys is in use.

Two new statements have been added to support KSDS:

- DELETE, enables you to delete a record from a VSAM file after a READ operation.

- REWRITE, enables you to replace a record in a VSAM file after a READ operation.

Reentrant Object Code (MVS and VM)

The compiler can create a reentrant version of the object-code portion of a program. When object code is reentrant (and placed in a reentrant area), multiple end-users can share a single copy, thereby saving execution-time storage.

Execution-Time Loading of Library Routines

The library has been restructured to allow more execution-time loading of library routines. This has multiple benefits:

- Reduces auxiliary storage requirements for load modules

- Speeds execution for users in compile-link-go mode

- In an MVS/XA environment, allows many library routines to reside above 16 megabytes, thus providing virtual-storage constraint relief.

(This new library design will not impact users who have Release 2 or Release 3 load modules that access the old reentrant I/O library (via IFYVRENT), and who do not want to relink. Maintenance is automatically provided, and relinking is necessary only if Release 4 function is desired.)

Automatic Precision Increase

This feature allows a user to selectively boost the precision of floating-point items in an existing program without recoding it. Single precision items can be made double, double can be made extended. Users merely recompile the program with a specified option (AUTODBL).

Faster Character Handling

Character assignment and comparison operations are now handled by in-line code, rather than by calls to the library. This speeds execution time. Error messages previously issued from the library, for conditions such as overlap detection and invalid character length, will no longer appear.

Improved Diagnostic Support

The following enhancements will allow easier program maintenance and debugging.

- MAP and XREF output can be formatted to fit a terminal screen.

- LIST output now gives ISNs, and XREF output now identifies variables referenced but not initialized.

- An explicit SDUMP compiler option is now available (previously, this was available only as an installation-wide default).
- SDUMP tables have been condensed and simplified, decreasing object module size. The symbol table size, however, remains the same.
- Execution-time error messages have been expanded to supply line numbers, ISNs, and offsets.

Improved I/O Support

The following improvements have been made to VS FORTRAN I/O statements:

- For sequential unformatted I/O, you can now use all record formats. Fixed, fixed blocked, undefined, variable, and variable spanned formats are supported.
- You can now use data initialization values in the character and double precision explicit-type statements.
- You can specify a character type unit designator for list-directed READ and WRITE statements. This allows you to do list-directed reads and writes to an internal file.
- The NUM parameter is now a valid control list parameter for the unformatted READ I/O statements for LANGLVL(77). The NUM parameter returns the number of bytes transferred.
- Several extensions have been made to the namelist READ and WRITE statements. You can now use the keywords UNIT and FMT. The unit designator for namelist I/O can be character type, so you can do namelist reads and writes to an internal file. The unit designator can also be an asterisk to represent an installation-dependent unit. You can now use a shortened form for reading and printing at LANGLVL(77).

Release 3.1, March 1984

VS FORTRAN Interactive Debug Support

When a VS FORTRAN program is executed, the user has a choice of two different execution options:

- DEBUG, which activates VS FORTRAN Interactive Debug immediately; and
- NODEBUG, the IBM default, which does not invoke VS FORTRAN Interactive Debug.

Note: The TEST compiler option is not necessary for VS FORTRAN Interactive Debug.

Release 3.0, March 1983

Character Data Type Handling

VS FORTRAN Release 3.0 provides for passing character length arguments in a manner that is not apparent to the user.

In addition:

- Character and noncharacter data types are allowed in the same common block.
- Character and noncharacter data types are allowed in an EQUIVALANCE relationship.
- The CHARLEN compiler option may be specified to set the maximum length of the character data type to a range of 1 through 32767. The default maximum length remains 500 characters, or whatever was set at installation time.
- The SC option has been removed because the character length is now passed in a manner that is not apparent to the user.

Debugging and Diagnostic Aids

- The TRMFLG compiler option may be specified to display a source statement in error on the SYSTERM data set, along with the diagnostic message.
- A symbolic dump of variables at abnormal termination can be obtained for modules *not* compiled with the NOSDUMP compiler option.
- A symbolic dump of variables in a module *not* compiled with the NOSDUMP option can be obtained on request by calling the SDUMP library routine.
- The SYM compiler option may be specified to produce SYM cards along with the object deck.
- The SRCFLG compiler option may be specified to insert diagnostic messages in the printed source listing.

INCLUDE Statement Improvement

- INCLUDE statements can be selectively activated during compilation.
- Blocked file support has been added to the INCLUDE facility.

Miscellaneous Changes

- OPEN, CLOSE, and INQUIRE parameters that are constants are checked at compile time.
- VS FORTRAN continues executing after transmission input/output errors have occurred.

- Formatting for a new direct-access data set has been provided for the OPEN statement.

- For direct-access I/O, the records of a file must be either all formatted or all unformatted, not mixed.

- Various service changes have been made.

Warning: Every program that has been compiled with versions of VS FORTRAN previous to Release 3.0, and that either references or defines a user subprogram that has character-type arguments or is itself of character type, must be recompiled with VS FORTRAN Release 3.0.

Contents

Part 1. Language Reference 1

Chapter 1. Introduction 3
Language 3
Compiler 3
Execution-Time Library 3
Valid and Invalid VS FORTRAN Programs 4

Chapter 2. VS FORTRAN Language 5
Language Definitions 5
Language Syntax 6
  Input Records 6
  Source Language Statements 7
  Source Statement Characters 10
  Names 11
  Statement Labels 13
  Keywords 13

Chapter 3. VS FORTRAN Data 15
Constants 15
  Arithmetic Constants 16
  Logical Constants 21
  Character Constants 21
  Hollerith Constants 22
  Hexadecimal Constants 23
Variables 24
  Variable Names 24
  Variable Types and Lengths 25
Array 28
  Subscripts 28
  Size and Type Declaration of an Array 30
Character Substrings 32

Chapter 4. VS FORTRAN Expressions 35
Evaluation of Expressions 35
Arithmetic Expressions 36
  Arithmetic Operators 36
  Rules for Constructing Arithmetic Expressions 37
  Use of Parentheses in Arithmetic Expressions 39
  Type and Length of the Result of Arithmetic Expressions 39
  Examples of Arithmetic Expressions 43
Character Expressions 44
  Use of Parentheses in Character Expressions 45

Relational Expressions 45
Logical Expressions 47
Logical Operators 48
Order of Computations in Logical Expressions 49
Use of Parentheses in Logical Expressions 51

Chapter 5. VS FORTRAN Statements 53
VS FORTRAN Statement Categories 53
Assignment Statements 54
Control Statements 54
DATA Statement 54
Debug Statements 54
Input/Output Statements 55
PROGRAM Statement 55
Specification Statements 55
Subprogram Statements 56
VS FORTRAN Compiler Directive Statements 57
Order of Statements in a Program Unit 57
VS FORTRAN Statement Descriptions 58
Arithmetic IF Statement 59
ASSIGN Statement 59
Assigned GO TO Statement 60
Assignment Statements 60
AT Statement 66
BACKSPACE Statement 67
BLOCK DATA Statement 69
Block IF Statement 70
CALL Statement 70
Character Type Statement 73
CLOSE Statement 74
Comments 76
COMMON Statement 77
Complex Type Statement 79
Computed GO TO Statement 79
CONTINUE Statement 79
DATA Statement 80
DEBUG Statement 82
DELETE Statement 87
DIMENSION Statement 87
DISPLAY Statement 88
DO Statement 89
Double Precision Type Statement 93
EJECT Statement 94
ELSE Statement 94
ELSE IF Statement 94
END Statement 94
END DEBUG Statement 95
ENDFILE Statement 96
END IF Statement 97
ENTRY Statement 97
EQUIVALENCE Statement 101
Explicit Type Statement 103
EXTERNAL Statement 108
FORMAT Statement 108

FUNCTION Statement 137
GO TO Statements 142
IF Statements 145
IMPLICIT Type Statement 151
INCLUDE Statement 153
INQUIRE Statement 154
INTRINSIC Statement 164
Logical IF Statement 165
Logical Type Statement 165
NAMELIST Statement 166
OPEN Statement 168
PARAMETER Statement 173
PAUSE Statement 175
PRINT Statement 175
PROGRAM Statement 177
READ Statements 178
REAL Type Statement 221
RETURN Statement 222
REWIND Statement 225
REWRITE Statement—Formatted with Keyed Access 227
REWRITE Statement—Unformatted with Keyed Access 230
SAVE Statement 232
Statement Function Statement 233
Statement Numbers 236
STOP Statement 237
SUBROUTINE Statement 238
TRACE OFF Statement 243
TRACE ON Statement 243
Unconditional GO TO 243
WAIT Statement 244
WRITE Statements 246

Chapter 6. VS FORTRAN Intrinsic Functions 285

Part 2. Library Reference 303

Chapter 7. Introduction 305

Chapter 8. Mathematical, Character, and Bit Subprograms 307
Explicitly Called Subprograms 307
Implicitly Called Subprograms 308

Chapter 9. Service and Utility Subroutines 313
Mathematical Exception Test Subprograms 313
DVCHK Subroutine 313
OVERFL Subroutine 313
Utility Subprograms 314
DUMP/PDUMP Subroutine 315
CDUMP/CPDUMP Subroutine 317
EXIT Subroutine 317
OPSYS Subroutine (VSE Only) 318
SDUMP SUBROUTINE 319
XUFLOW SUBROUTINE 322

Chapter 10. Extended Error Handling Subroutines and Error Option Table 323
Extended Error Handling 323
Error Handling Subroutines 324
ERRMON Subroutine 324
ERRSAV Subroutine 325
ERRSET Subroutine 325
ERRSTR Subroutine 329
ERRTRA Subroutine 329
Error Option Table 330

Appendix A. Source Language (FIPS) Flagger 347
Items Flagged for Full ANS Language 347
Global Items Flagged 347
Statements Flagged 347
Execution-Time Cautions 349

Appendix B. IBM and ANS FORTRAN Features 351
New ANS FORTRAN 1977 Features 351
General Features 351
New Statements 353
New Features in Old Statements 353
Old IBM Extensions Now in ANS FORTRAN 1977 356
IBM Extensions Not in ANS FORTRAN 1977 357
LANGLVL(66) Features Not in LANGLVL(77) 358

Appendix C. EBCDIC and ISCII/ASCII Codes 361

Appendix D. Algorithms for Library Mathematical Functions 369
Control of Program Exceptions in Mathematical Functions 370
Explicitly Called Subprograms 372
Absolute Value Subprograms 372
Arcsine and Arccosine Subprograms 373
Arctangent Subprograms 376
Error Functions Subprograms 379
Exponential Subprograms 384
Gamma and Log Gamma Subprograms 387
Hyperbolic Sine and Cosine Subprograms 390
Hyperbolic Tangent Subprograms 392
Logarithmic Subprograms (Common and Natural) 393
Sine and Cosine Subprograms 398
Square Root Subprograms 401
Tangent and Cotangent Subprograms 405
Implicitly Called Subprograms 409
Complex Multiply and Divide Subprograms 409
Complex Exponentiation 411
Exponentiation of a Real Base to a Real Power 412
Exponentiation of a Real Base to an Integer Power 414
Exponentiation of an Integer Base to an Integer Power 416
Exponentiation of a Base 2 Argument to a Real Power 416

Appendix E. Storage Estimates 417

Appendix F. Accuracy Statistics 425

Appendix G. Assembler Language Information 433
Library Availability 433
Calling Sequences 434
  Assembler Language Calling Sequence 435
Supplying Correct Parameters 435
  Mathematical Subprogram Results 436
Space Considerations 436
Initializing the Execution Environment 436

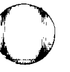
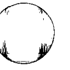
Appendix H. Sample Storage Printouts 445
Output from Symbolic Dumps 447
Output Format 447
  Scalar Noncharacter 448
  Scalar Character 448
  Array 449
  Control Flow Information 450
  I/O Unit Information 451
  I/O Unit Status Information 452
Examples of Sample Programs and Symbolic Dump Output 452

Appendix I. Library Procedures and Messages 463
Library Interruption Procedures 463
Library Error Procedures 463
Library Messages 464
  Program-Interrupt Messages 465
Execution Error Messages 468
  Operator Messages 531

Appendix J. Library Module Names 533

Glossary 545

Index 551

Figures

1. Example of Fixed-Form Source Statements 8
2. Example of Free-Form Source Statements 10
3. Source Statement Characters (VS FORTRAN Character Set) 11
4. Data Types and Valid Lengths 26
5. Examples of Arithmetic Expressions 36
6. Arithmetic Operators 37
7. Hierarchy of Arithmetic Operations 38
8. Type and Length of Result Where the First Operand Is Integer 40
9. Type and Length of Result Where the First Operand Is Real 41
10. Type and Length of Result Where the First Operand Is Complex 42
11. Character Operator 44
12. Relational Operators 45
13. Logical Operators 48
14. Hierarchy of Operations Involving Arithmetic Operators 49
15. Hierarchy of Operations Involving Character Operators 50
16. Type and Length of the Result of Logical Operations 52
17. Order of Statements and Comment Lines 58
18. Conversion Rules for the Arithmetic Assignment Statement a=b, Where Type of b Is Integer or Real 62
19. Conversion Rules for the Arithmetic Assignment Statement a=b, Where Type of b Is Complex 63
20. Field Widths Needed for Data Types of Various Lengths 276
21. Logarithmic and Exponential Functions 288
22. Trigonometric Functions 289
23. Hyperbolic Functions 291
24. Miscellaneous Mathematical Functions 292
25. Conversion and Maximum/Minimum Functions 296
26. Character Manipulation Functions 299
27. Bit Manipulation Functions 300
28. Generic Names for Intrinsic Functions 301
29. Implicitly Called Mathematical Subprograms 309
30. Implicitly Called Character Subprograms 310
31. Implicitly Called Service Subprograms 311
32. Exponentiation with Integer Base and Exponent 311
33. Exponentiation with Real Base and Integer Exponent 311
34. Exponentiation with Real Base and Exponent 312
35. Exponentiation with Complex Base and Integer Exponent 312
36. Option Table Preface Entry 330
37. Error Option Table Entry 331
38. Option Table Default Values 333
39. IOSTAT and ERR Parameters Honored for I/O Errors 334
40. Corrective Action after Error 336
41. Corrective Action after Program Interrupt 340
42. Corrective Action after Mathematical Subroutine Error 342

43. Mathematical Subprogram Storage Estimates 417
44. Service Subprogram Storage Estimates 421
45. Character Subprogram Storage Estimates 421
46. Bit Subprogram Storage Estimates 422
47. Table of Storage Estimates for Library Execution-Time Routines 424
48. Accuracy Figures 427
49. Explicitly Called Mathematical Subprogram Assembler Information 438
50. Implicitly Called Mathematical Subprogram Assembler Information 440
51. Implicitly Called Character Subprogram Assembler Information 440
52. Service Subprogram Assembler Information 441
53. Explicitly Called Bit Function Assembler Information 441
54. General Assembler Language Calling Sequence 442
55. Examples of Assembler Language Calling Sequences 443
56. Sample Storage Printout for DUMP/PDUMP and CDUMP/CPDUMP 446
57. Sample Storage Printout for SDUMP 447
58. Entry Names for Library Modules 533
59. Reentrant Library Module Names 543

Part 1. Language Reference

The following topics are discussed in Part 1:

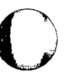
Introduction

VS FORTRAN Language

VS FORTRAN Data

VS FORTRAN Expressions

VS FORTRAN Statements



Chapter 1. Introduction

IBM VS FORTRAN consists of a language, a compiler, and an execution-time library of subprograms.

Language

The VS FORTRAN language consists of a set of characters, conventions, and rules that are used to convey information to the compiler. The basis of the VS FORTRAN language is a *statement* containing combinations of element names, operators, constants, and words (keywords) whose meaning is predefined to the compiler.

The VS FORTRAN language is best suited to applications that involve mathematical computations and other manipulation of arithmetic data.

Compiler

In the process of compilation, a program called the VS FORTRAN compiler analyzes the source program statements and translates them into a machine language program called the object program, which can be combined with library routines to form a program suitable for execution. When the VS FORTRAN compiler detects errors in the source program, it produces appropriate diagnostic messages.

The VS FORTRAN compiler operates under the control of an operating system that provides it with input, output, and other services. Object programs generated by the VS FORTRAN compiler also operate under operating system control and depend on it for similar services.

Execution-Time Library

The VS FORTRAN execution-time library consists of subroutines and functions supplied as part of the product. For complete information on the library, see "Part 2. Library Reference" on page 303 For a description of the intrinsic functions and source subroutines to which the user may refer directly in VS FORTRAN statements, see "Explicitly Called Subprograms" on page 307. For a discussion of extended error handling subroutines, see Chapter 10, "Extended Error Handling Subroutines and Error Option Table" on page 323.

Subroutines and functions to furnish any commonly used code sequences can be compiled and added to an execution-time library by the user. When written in VS FORTRAN, these can be structured as function, subroutine, or block data subprograms. Other source languages can be used if the subroutines are accessible by VS FORTRAN calls. User subroutines may reside in the supplied library data set or in a private data set called at load or link-edit time.

Valid and Invalid VS FORTRAN Programs

This manual defines the rules (that is, the syntax, semantics, and restrictions) applicable for writing valid VS FORTRAN programs, either for the 1978 Standard or for the 1978 Standard plus IBM extensions. Most violations of the VS FORTRAN language rules are diagnosed by the compiler; however, some syntactic and semantic combinations are not diagnosed, some because they are detectable only at execution time, others for performance reasons. VS FORTRAN programs that contain these undiagnosed combinations are invalid VS FORTRAN programs, whether or not they execute as expected.

Chapter 2. VS FORTRAN Language

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

Data Consists of constants, variables, and arrays. See Chapter 3, "VS FORTRAN Data" on page 15.

Expressions Executable sets of arithmetic, character, logical, or relational data. See Chapter 4, "VS FORTRAN Expressions" on page 35.

Statements Combinations of data and expressions. See "VS FORTRAN Statement Descriptions" on page 58.

Language Definitions

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

Main program. A program unit, required for execution, that can call other program units but cannot be called by them. A main program does *not* have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. The main program is the first program unit to receive control at execution time.

Subprogram. A program unit that is invoked by another program unit in the same program. In FORTRAN, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

Procedure. A sequenced set of statements that may be used at one or more points in one or more computer programs, and that usually is given one or more input parameters and returns one or more output parameters. A procedure consists of subroutines, function subprograms, and intrinsic functions.

Intrinsic function. A function, supplied by VS FORTRAN, that performs mathematical or character operations. (See "INTRINSIC Statement" on page 164.)

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

Executable program. A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures, or both.

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

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

Preconnected file. A unit or file that was defined at installation time. However, a preconnected file does not exist for a program if the file is not defined by a FILEDEF command or by job control statements.

Program unit. A sequence of statements constituting a main program or subprogram.

Additional definitions can be found in the "Glossary" on page 545.

Language Syntax

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

Source language statements
Source statement characters
Names
Statement labels
Keywords

Input Records

VS FORTRAN accepts source input in either of two formats:

- Fixed-form input format

IBM Extension

- Free-form input format

End of IBM Extension

A program unit must be written in either fixed form or free form, not both. For a detailed description of the use and implementation of the two formats, see the FIXED | FREE compiler option in the *VS FORTRAN Programming Guide.*

The VS FORTRAN compiler receives its input in fixed-length, 80-byte records. Each record is equivalent to one 80-column card or one 80-character input line on a terminal.

Source Language Statements

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

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

- Statements, which may occupy one or more input lines, provide the information needed by the VS FORTRAN 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; otherwise, they are not processed by the compiler. 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 on the source program listing to make the program more readable.

Fixed-Form Input Format

The statements and comments of a VS FORTRAN 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. Comments may appear anywhere in columns 2 through 72. Comment lines may appear anywhere in a program unit before the END statement. (Comment lines may precede a continuation line.) Blank lines may appear anywhere in a program unit and are processed as comment lines.

- Statement Text

  The text of a fixed-form statement is written in columns 7 through 72 on 1 to 20 lines. The statement text may continue on as many as 19 continuation lines. Multiple statements per line are not allowed. Every statement in a program unit may have a label in columns 1 to 5. Column 6 is used to distinguish between initial and continuation lines. Columns 73 through 80 are not part of the statement and may be used for identification. A statement is terminated by another statement or by the end of the input.

- Statement Labels

  The statement label consists of from 1 to 5 decimal digits anywhere in columns 1 to 5 in the initial line of a statement. The value must not be zero. Values of labels do not affect the order in which statements are compiled or executed. Each label must be unique; that is, the same label must not be given to more than one statement within a program unit.

- Initial Line

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

- Continuation Lines

  A statement that is not complete on the initial line may continue in columns 7 through 72 on as many as 19 continuation lines. A continuation line must have a character that is not blank or zero in column 6.

IBM Extension

VS FORTRAN allows columns 1 through 5 on a continuation line to contain characters, but they are ignored. (Note that a C or an asterisk (*) in column 1 will cause the line to be treated as a comment.)

End of IBM Extension

- Identification

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

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

Figure 1 illustrates fixed-form source statements.

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

Figure 1. Example of Fixed-Form Source Statements

Free-Form Input Format

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

- Comments

  A comment line begins with a quotation mark (") in column 1. A comment line must not follow a continued line, and cannot itself be continued. Blank lines are not allowed with free-form input.

- Statement Text

  The text of free-form statements is entered in 80 columns on 1 to 20 lines. The first character of a statement (after a label, if any) must be alphabetic. Multiple statements per line are not allowed. The statement text may continue on as many as 19 succeeding continuation lines. A continued line has a minus sign (-) as the final (rightmost) character on the line. The line following a continued line is a continuation line. A statement is terminated by an initial or continuation line that does not end with a minus sign. Columns 73 through 80, which may be used for identification in fixed-form statements, are considered part of the statement text in free form.

- Statement Labels

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

- Initial Line

  The initial line of a statement may have a label. The first character of the statement text must be alphabetic. If a statement does not have a label, then the statement text must begin on the initial line. (Blank lines are not allowed.)

- Continued Lines

  The text of any statement, except the END statement, may continue on the following line. A line to be continued is indicated by terminating the line with a minus sign (-). A comment line cannot be continued.

- Preserving a Minus Sign

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

- Continuation Lines

  A continuation line is a line following a continued line. The statement text may start in any position. Up to 19 continuation lines are permitted in a single statement.

- Maximum Statement Length

  The maximum length of a free-form source statement is 1320 characters, excluding the continuation characters and the statement label. Blank characters are counted in the total number of characters. Any blank characters after the continuation characters are not counted.

Figure 2 illustrates free-form source statements.

```
Column:    1
           --------------------
           "SAMPLE TEXT
             .
             .
             .
           10D=010.5
           GO TO 56
           150 A=B+C*(D+E**F+-
           G+H-2.*(G+P))
           C=3.
             .
             .
             .
```

Figure 2. Example of Free-Form Source Statements

End of IBM Extension

Source Statement Characters

The characters listed in Figure 3 on page 11 constitute the set of characters acceptable in a VS FORTRAN program. The set is commonly referred to as the VS FORTRAN character set.

A special character may be an operator (or part of an operator), part of a constant, or have some other special meaning. The interpretation is implied by the context.

The special characters shown in Figure 3 are listed in their correct collating sequence. (The complete collating sequence can be found in Appendix C, "EBCDIC and ISCII/ASCII Codes" on page 361.)

SPECIAL CHARACTERS		LETTERS		DIGITS
	blank	A	O	0
.	period	B	P	1
(	left parenthesis	C	Q	2
+	plus sign	D	R	3
$	currency sign	E	S	4
*	asterisk	F	T	5
)	right parenthesis	G	U	6
–	minus sign	H	V	7
/	slash	I	W	8
,	comma	J	X	9
:	colon	K	Y	
'	apostrophe	L	Z	
=	equal sign	M		
"	quotation mark	N	$	

Figure 3. Source Statement Characters (VS FORTRAN Character Set)

Names

Names (referred to as "symbolic names" in old FORTRAN publications) can be assigned to the elements of a program unit.

Definition

Name—A string of 1 through 6 letters (A,B,...,Z) or digits (0,1,...,9), the first of which must be a letter.

IBM Extension

With this compiler, the currency symbol ($) is treated as a letter when used in a name. Therefore, the currency symbol ($) can be used as the first character in a name.

End of IBM Extension

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

- An array and the elements of that array (see "Array" on page 28)
- A variable (see "Variables" on page 24)
- A constant (See "PARAMETER Statement" on page 173)

- A main program (see "PROGRAM Statement" on page 177)

- A statement function (see "Statement Function Statement" on page 233)

- An intrinsic function (see "INTRINSIC Statement" on page 164)

- A function subprogram (see "FUNCTION Statement" on page 137)

- A subroutine subprogram (see "SUBROUTINE Statement" on page 238)

- A block data subprogram (see "BLOCK DATA Statement" on page 69)

- A common block (see "COMMON Statement" on page 77)

- An external user-supplied subprogram that cannot be classified by its usage in that program unit as either a subroutine or function subprogram name (see "EXTERNAL Statement" on page 108)

- A NAMELIST (see "READ Statement—NAMELIST with External Devices" on page 216 and "WRITE Statement—NAMELIST with External Devices" on page 280)

A name that identifies a constant, variable, array, external function, or statement function also identifies its data type. The name may be specified in a specification statement (see "Specification Statements" on page 55). If the name does not appear in such a statement, the type is implied by the first letter of the name. A first letter of I through N implies integer type, and any other letter (or the currency symbol) implies real type, unless an IMPLICIT statement is used to change the default type.

Names are either global or local. Global names are recognized both internal to and external to a program unit. Local names are recognized internal to the program unit where they are referenced.

- Classes of global names are:

  - Common block

  - External function

  - Subroutine

  - Main program

  - Block data subprogram

- Classes of local names are:

  - Array

  - Variable

  - Constant

- Statement function
- Intrinsic function
- Dummy procedure

Names must be unique within a class in a program unit and can identify elements of only one class, 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.

The name of a main program, subroutine, common block, NAMELIST, or block data subprogram has no type. A generic function name has no predetermined type; it assumes a type dependent upon the type of its argument(s).

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

Statement Labels

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

A statement to which control is passed

The end of a sequence of statements which are to be executed repeatedly

A formatting statement

A statement label is a sequence of from 1 to 5 decimal digits, one of which must be nonzero. It can be written in either fixed form or free form. See "Statement Numbers" on page 236.

Keywords

Keywords identify VS FORTRAN-supplied procedures (intrinsic functions) that can be used as part of any program. These procedures are mathematical functions and service subroutines, which are supplied to save programmers time. See Appendix B, "IBM and ANS FORTRAN Features" on page 351.

A keyword is a specified sequence of characters. Whether a particular sequence of characters identifies a keyword or a name is implied by the context. There is no sequence of characters that is reserved in all contexts.

Chapter 3. VS FORTRAN Data

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

- Constants
- Variables
- Arrays

Note: These are not to be confused with *data types*. Data types correspond to the the five types of variables, as discussed under "Variable Types and Lengths" on page 25.

Constants

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

- **Arithmetic** constants specify decimal values. There are three 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 a string of alphameric and/or special characters enclosed in apostrophes.

IBM Extension

- **Hollerith** constants are used only in FORMAT statements.
- **Hexadecimal** constants are used only as data initialization values of any type of variable.

End of IBM Extension

The PARAMETER statement allows a constant to be given a name. (See "PARAMETER Statement" on page 173.)

Arithmetic Constants

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

An unsigned constant is a constant with no leading sign. A signed constant is a constant with a leading plus or minus sign. An optionally signed constant is a constant that may be either signed or unsigned. Only integer and real constants may be optionally signed.

Integer Constants

Definition

Integer Constant—A string of decimal digits containing no decimal point and expressing a whole number. It occupies 4 bytes of storage.

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

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

Valid Integer Constants:

```
0

91

173

-2147483647
```

Invalid Integer Constants:

```
27.             Contains a decimal point.

3145903612      Exceeds the maximum magnitude.

5,396           Contains an embedded comma.

-2147483648     Exceeds the maximum magnitude,
                even though it fits into 4 bytes.
```

Real Constants

Definition

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

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 also be explicitly specified by appending an exponent to a basic real constant or an integer constant. The standard exponents consist of the letters E and D.

IBM Extension

This compiler also allows the letter Q as an exponent.

End of IBM Extension

An exponent is followed by a signed or unsigned 1- or 2-digit 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.

IBM Extension

The letter Q specifies a constant of length 16 and occupies 16 bytes of storage.

End of IBM Extension

Magnitude: 0 or 16^{-65} (approximately 10^{-78}) through 16^{63} (approximately 10^{75})

Precision: (Four bytes) 6 hexadecimal digits (approximately 6 decimal digits)

(Eight bytes) 14 hexadecimal digits (approximately 15 decimal digits)

IBM Extension

(Sixteen bytes) 28 hexadecimal digits (approximately 32 decimal digits)

End of IBM Extension

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

product of a basic real constant or integer constant and 10 raised to a desired power.

Valid Real Constants (Four Bytes):

```
+0.

-999.9999

7.0E+0              That is, 7.0 x 10^0 = 7.0

9761.25E+1          That is, 9761.25 x 10^1 = 97612.5

7.E3

7.0E3               That is, 7.0 x 10^3 = 7000.0

7.0E+03

7E-03               That is, 7.0 x 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):

```
1234567890123456.D-73   Equivalent to .1234567890123456x10^-57

7.9D03

7.9D+03                 That is, 7.9 x 10^3 = 7900.0

7.9D+3

7.9D0                   That is, 7.9 x 10^0 = 7.9

7D03                    That is, 7.0 x 10^3 = 7000.0
```

IBM Extension

Valid Real Constants (Sixteen Bytes):

```
.234523453456456734565678Q+43

5.001Q08
```

End of IBM Extension

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 1.0×10^{0}.
1.2E+113	Too many digits in the exponent.
23.5D+97	Magnitude outside the allowable range, that is, $23.5 \times 10^{97}. > 16^{63}$.
21.3D-99	Magnitude outside the allowable range, that is, $21.3 \times 10^{-99} < 16^{-65}$.

IBM Extension

88.63215748Q123	Too many digits in the exponent

End of IBM Extension

Complex Constants

Definition

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

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

IBM Extension

If the constants of the ordered pair representing the complex constant differ in precision, the constant of lower precision is converted to a constant of the higher precision.

For example, if one constant is real and the other is double precision, real is converted to double precision.

If the constants differ in type, the integer constant is converted to a real constant of the same precision as the original real constant.

For example, if one constant is integer and the other is double precision, the integer constant is converted to a double precision constant.

End of IBM Extension

Valid Complex Constants (i = square root of -1):

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

IBM Extension

(-5.0E+03,.16D+02)	Has the value -5000.+16.0i; both parts are double precision.
(4.7D+2,1.973614D4)	Has the value 470.+19736.14i.
(47D+2,38D+3)	Has the value 4700.+38000.i.
(1234.345456567678Q59,-1.0Q-5)	
(45Q6,6E45)	Both parts are real (16 bytes long).

End of IBM Extension

Invalid Complex Constants:

(A, 3.7)	Real part is not a constant.

IBM Extension

(.0009Q-1,7643.Q+1199)	Too many digits in the exponent of the imaginary part.
(49.76, .015D+92)	Magnitude of imaginary part is outside of allowable range.

End of IBM Extension

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.

IBM Extension

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

End of IBM Extension

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 47.)

Character Constants

Definition

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

The delimiting apostrophes are not part of the data represented by the constant. An apostrophe within the character data is represented by two consecutive apostrophes, with no intervening blanks. In a character constant, blanks embedded between the delimiting apostrophes are significant. The length of a character constant must be greater than zero. Each character requires one byte of storage.

The maximum length of a character constant depends upon the circumstance of use and, where significant, the number of continuation cards. The number of continuation cards are as follows:

- Data initialization (maximum of 1310)
- Assignment statement (maximum of 1316)
- Argument of a call (maximum of 1311)

- Input or output statement (maximum of 1309)
- FORMAT statement (maximum of 1310)
- PARAMETER statement (maximum of 255)
- PAUSE or STOP statement (maximum of 72)

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

- A character expression
- An assignment statement
- The argument list of a CALL statement or function reference
- An input or output statement
- A FORMAT statement
- A PARAMETER statement
- A PAUSE or STOP statement

Valid Character Constants:	Length:
`'DATA'`	4
`'X-COORDINATE     Y-COORDINATE     Z-COORDINATE'`	44
`'3.14'`	4
`'DON''T'`	5

IBM Extension

Hollerith Constants

Definition

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

Each character requires one byte of storage.

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

Valid Hollerith Constants:

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

6H DON'T
```

Hexadecimal Constants

Definition

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

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

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

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

Valid Hexadecimal Constants:

`Z1C49A2F1` represents the bit string:

```
000111000100100110100010111110001
```

`ZBADFADE` represents the bit string:

```
00001011101011011111101011011110
```

where the first 4 zero bits are implied because an odd number of hexadecimal digits is written.

The maximum number of digits allowed in a hexadecimal constant depends upon the length specification of the variable being initialized (see "Variable Types and Lengths" on page 25). The following list shows the maximum number of digits for each length specification:

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

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

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

End of IBM Extension

Variables

A VS FORTRAN variable is a data item, identified by a name, that occupies a storage area, except possibly in situations involving error or interruption handling, where normal program flow is asynchronously interrupted. The value represented by the name is always the current value stored in the area.

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

Variable Names

VS FORTRAN variable names must follow the rules governing element names. (See "Names" on page 11.) The use of meaningful variable names can aid in documenting a program.

Valid Variable Names:

B292S

RATE

IBM Extension

$VAR

End of IBM Extension

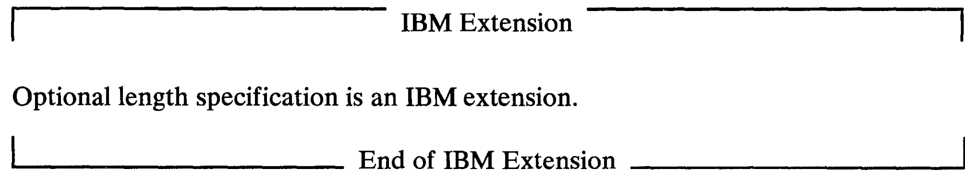
Invalid Variable Names:

B292704	Contains more than six characters.
4ARRAY	First character is not alphabetic.
SI.X	Contains a special character.

Variable Types and Lengths

The type of a variable corresponds to the type of data the variable represents. (See Figure 4 on page 26.) Thus, an integer variable must represent integer data, a real variable must represent real data, and so on. There is no variable type associated with hexadecimal data; this type of data is identified by a name of one of the other types. There is no variable type associated with statement numbers; integer variables that contain the statement number of an executable statement or a FORMAT statement are not considered to contain an integer variable. (See "ASSIGN Statement" on page 59.)

For every type of variable data, there is a corresponding length specification that determines the number of bytes that are reserved.



IBM Extension

Optional length specification is an IBM extension.

End of IBM Extension

Figure 4 shows each data type with its associated storage length and standard length.

DATA TYPE	VALID STORAGE LENGTHS	DEFAULT LENGTH
Integer	[2 or] 4	4
Real	4 [, 8 or 16]	4
Double Precision	8	8
Complex	8 [, 16 or 32]	8
Character	1 through 32767	1
Logical	[1 or] 4	4

Figure 4. Data Types and Valid Lengths

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

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

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

IBM Extension

VS FORTRAN accepts:

- INTEGER*2 to indicate 2 bytes and INTEGER*4 as an alternative to INTEGER, to indicate 4 bytes;
- REAL*4 as an alternative to REAL, to indicate 4 bytes;
- REAL*8 as an alternative to DOUBLE PRECISION, to indicate 8 bytes;

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

End of IBM Extension

Type Declaration by the Predefined Specification

The predefined specification is a convention used to specify variables as integer or real as follows:

- 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.

IBM Extension

- If the first character of the variable name is a currency symbol ($), the variable is real of length 4.

End of IBM Extension

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

Type Declaration by the IMPLICIT Statement

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

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

The IMPLICIT statement is discussed in "IMPLICIT Type Statement" on page 151.

Type Declaration by Explicit Specification Statements

Explicit specification statements differ from the first two ways of specifying the type of a variable, in that an explicit specification statement declares the type of a particular variable by its *name* rather than by a group of variable names beginning with a particular *letter*, as specified in Figure 3 on page 11. Explicit type statements override IMPLICIT statements and predefined specifications.

Explicit specification statements are discussed in "Explicit Type Statement" on page 103.

Array

An array is an ordered and structured sequence of data items. The data items that make up the array are 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. A particular element in the array is identified by the array name and its position in the array. All elements of an array have the same type and length.

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

Before an array element has been assigned a value, its content is undefined, and the array element should not be referred to before assigning it a value.

Subscripts

A subscript is a quantity (or a set of subscript expressions separated by commas) that is 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 with whose name the subscript is associated. 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. (See Chapter 4, "VS FORTRAN Expressions" on page 35 for additional information and restrictions.)

1. Subscript expressions may contain arithmetic expressions that use any of the arithmetic operators: +, -, *, /, **.

2. Subscript expressions may contain function references that do not change any other value in the same statement.

3. Subscript expressions may contain array elements.

IBM Extension

4. Mixed-mode expressions (integer and real only) within a subscript are evaluated according to normal FORTRAN rules. If the evaluated expression is real, it is converted to integer by truncation.

End of IBM Extension

5. The evaluated result of a subscript expression must always be greater than or equal to the corresponding lower dimension bound and must not exceed the corresponding upper dimension bound (See "Size and Type Declaration of an Array" on page 30.)

Valid Array Elements:

```
ARRAY (IHOLD)

NEXT (19)

MATRIX (I-5)
```

IBM Extension

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

End of IBM Extension

```
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 three dimensions:

DIMENSION C(1:3,1:2,1:4)

```
C(1,1,1) C(2,1,1) C(3,1,1) C(1,2,1) C(2,2,1) C(3,2,1)
C(1,1,2) C(2,1,2) C(3,1,2) C(1,2,2) C(2,2,2) C(3,2,2)
C(1,1,3) C(2,1,3) C(3,1,3) C(1,2,3) C(2,2,3) C(3,2,3)
C(1,1,4) C(2,1,4) C(3,1,4) C(1,2,4) C(2,2,4) C(3,2,4)
```

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 (*e*1) and a required upper bound (*e*2) in the form:

Syntax

name ([*e*1:] *e*2)

name

is an array name.

where:

***e*1**

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

***e*2**

is the upper dimension bound and must always be specified.

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

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

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

DIMENSION IARY(3)
```

The upper and lower bounds (*e*1 and *e*2) are arithmetic expressions in which all constants and variables are of integer type.

- If the array name is an **actual** argument, the expressions can contain only constants or names of constants of integer type.
- The value of the lower bound may be positive, negative, or zero. It is assumed to be 1, if it is not specified.
- A maximum of seven dimensions is permitted. The size of each dimension is equal to the difference between the upper and lower bounds plus 1. If the value of the lower dimension bound is 1, the size of the dimension is equal to the value of its upper bound.
- Function or array element references are not allowed in dimension bound expressions.
- The value of the upper bound must be greater than or equal to the value of the lower bound. An upper dimension bound of an asterisk is always greater than or equal to the lower dimension bound.

- If the array name is a **dummy** argument and is in a subprogram, the expressions can also contain:
  - Integer variables that are also dummy arguments
  - Expressions that contain:
    - Signed or unsigned integer constants
    - Names of integer constants
    - Variables that are dummy arguments or appear in a common block in that subprogram
- The upper dimension bound of the last dimension of a dummy array name can be an asterisk. In this case, the dummy array is called an assumed-size array.

Size information must be given for all arrays in a VS FORTRAN program, so that an appropriate amount of storage may be reserved. Declaration of this information is made by a DIMENSION statement, by a COMMON statement, or by one of the explicit type specification statements. These statements are discussed in detail, in alphabetic sequence, in "VS FORTRAN Statement Descriptions."

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

Object-Time Dimensions

If a dummy argument array is used in a function or subroutine subprogram, the absolute dimensions of the array do not have to be explicitly declared in the subprogram by constants. Instead, the array declarators appearing in an explicit specification statement or DIMENSION statement in the subprogram may contain dummy arguments or variables in the common block that are integer variables of length 4, to specify the size of the array. When the subprogram is called, these integer variables receive their values from the actual arguments in the calling program reference or from the common block. Thus, the dimensions of a dummy array appearing in a subprogram may change each time the subprogram is called. This is called an "adjustable array" or an "object-time dimension array."

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

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

Character arrays are specified in the same manner as other data types. (See "DIMENSION Statement" on page 87 and "Explicit Type Statement" on page 103.) The length of each array element is either the standard length of 1 or may be declared larger with a type or IMPLICIT statement. Each character array element is treated as a single entity. Portions of an array element can be accessed through substring notation.

Character Substrings

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

The form of a substring reference is:

Syntax

a(*e*1:*e*2)

a

is a character variable name or a subscripted character array name (see "Array" on page 28).

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 *e*1 is omitted, a value of one is implied for *e*1. If *e*2 is omitted, a value equal to the length of the character variable or array element is implied for *e*2. Both *e*1 and *e*2 may be omitted; for example, the form *v*(:) is equivalent to *v*.

The value of *e*1 specifies the leftmost character position and the value of *e*2 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 *e*1 and *e*2 must be integer, positive, and nonzero.
- The value of *e*1 must be less than or equal to the value of *e*2.
- The values of *e*1 and *e*2 must be less than or equal to the number of characters contained in the corresponding variable name or array element.

Example 1:

Given the following statements:

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

then

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

Example 2:

Given the following statements:

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

then SUBSTG has the value XYVWZ.

Chapter 4. VS FORTRAN Expressions

VS FORTRAN provides four kinds of expressions: arithmetic, character, relational, and logical.

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

Evaluation of Expressions

VS FORTRAN expressions are evaluated according to the following rules:

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

  In an expression, an integer operand must be defined with an integer value, rather than a statement number. (See "ASSIGN Statement" on page 59.) If a character string or a substring is referred to, all of 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. The execution of a function reference in a statement 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.

- 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 primary, which may be a single constant, name of a constant, variable, array element, function reference, or another expression enclosed in parentheses. The primary may be either integer, real, or complex.

In an expression consisting of a single primary, the type of the primary is the type of the expression. Examples of arithmetic expressions are shown in Figure 5.

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

Figure 5. Examples of Arithmetic Expressions

Arithmetic Operators

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

The arithmetic operators are shown in Figure 6.

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

Figure 6. Arithmetic Operators

Rules for Constructing Arithmetic Expressions

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

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

  ```
  AB
  ```

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

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

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

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

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

  The expression A*-B could be written correctly as

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

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

- Order of Computation

  In the evaluation of expressions, priority of the operations is shown in Figure 7.

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

Figure 7. Hierarchy of Arithmetic Operations

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

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 8, Figure 9, and Figure 10 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.

Figure 8 shows the type and length of the result of adding, subtracting, multiplying, or dividing when the first operand is an **integer**.

Figure 9 shows the type and length of the result of adding, subtracting, multiplying, or dividing when the first operand is **real**.

Figure 10 shows the type and length of the result of adding, subtracting, multiplying, or dividing when the first operand is **complex**.

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

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

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

FIRST OPERAND / SECOND OPERAND	Integer (2)	Integer (4)
Integer (2)	Integer (2)	Integer (4)
Integer (4)	Integer (4)	Integer (4)
Real (4)	Real (4)	Real (4)
Real (8)	Real (8)	Real (8)
Real (16)	Real (16)	Real (16)
Complex (8)	Complex (8)	Complex (8)
Complex (16)	Complex (16)	Complex (16)
Complex (32)	Complex (32)	Complex (32)

Figure 8. Type and Length of Result Where the First Operand Is Integer

FIRST OPERAND / SECOND OPERAND	Real (4)	Real (8)	Real (16)
Integer (2)	Real (4)	Real (8)	Real (16)
Integer (4)	Real (4)	Real (8)	Real (16)
Real (4)	Real (4)	Real (8)	Real (16)
Real (8)	Real (8)	Real (8)	Real (16)
Real (16)	Real (16)	Real (16)	Real (16)
Complex (8)	Complex (8)	Complex (16)	Complex (32)
Complex (16)	Complex (16)	Complex (16)	Complex (32)
Complex (32)	Complex (32)	Complex (32)	Complex (32)

Figure 9. Type and Length of Result Where the First Operand Is Real

SECOND OPERAND \ FIRST OPERAND	Complex (8)	Complex (16)	Complex (32)
Integer (2)	Complex (8)	Complex (16)	Complex (32)
Integer (4)	Complex (8)	Complex (16)	Complex (32)
Real (4)	Complex (8)	Complex (16)	Complex (32)
Real (8)	Complex (16)	Complex (16)	Complex (32)
Real (16)	Complex (32)	Complex (32)	Complex (32)
Complex (8)	Complex (8)	Complex (16)	Complex (32)
Complex (16)	Complex (16)	Complex (16)	Complex (32)
Complex (32)	Complex (32)	Complex (32)	Complex (32)

Figure 10. Type and Length of Result Where the First Operand Is Complex

Examples of Arithmetic Expressions

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

NAME	TYPE	LENGTH
I, J, K	Integer variables	4 , 2, 2
C	Real variable	4
D	Complex variable	16

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

Subexpression **Type and Length**

```
I*J    (Call the result X)        Integer of length 4
C**K   (Call the result Y)        Real of length 4
X/Y    (Call the result Z)        Real of length 4
```

(X is converted to real of length 4 before division is performed.)

IBM Extension

```
Z+D                                Complex of length 16
```

(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. Then the real part of W is 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.

End of IBM Extension

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

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

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

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

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

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

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

Character Expressions

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

The character operator is shown in Figure 11.

Character Operator	Definition
//	Concatenation

Figure 11. Character Operator

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

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

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 103.

Use of Parentheses in Character Expressions

Parentheses have no effect on the value of a character expression. For example, 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.'

Valid Character Expressions:

Substring:

```
ST1311(I) = CVAR1(:I)
```

Function Reference:

```
ST1314(IVAR1) = CHAR(IVAR1)
```

Relational Expressions

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

The six relational operators are shown in Figure 12.

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

Figure 12. Relational Operators

Relational operators:

- Express a condition that can be either true or false.
- May be used to compare two arithmetic expressions (except complex) or two character expressions. Only the .EQ. and .NE. operators may be used to compare an arithmetic expression with a complex expression. If the two arithmetic expressions being compared are not of the same type or length, they are converted following the rules indicated in Figure 8, Figure 9, and Figure 10.

- In comparisons of arithmetic expressions to character expressions or vice versa are not allowed.

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 3 and in Appendix C, "EBCDIC and ISCII/ASCII Codes."

Examples:

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

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

Then the following examples illustrate valid and invalid relational expressions.

Valid Relational Expressions:

```
E .LT. I

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

.5 .GE. (.9*ROOT)

E .EQ. 27.3E+05

 CHAR .EQ. 'ABCDEFGH'

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

Invalid Relational Expressions:

`C.GE.(2.7,5.9E3)`	Complex quantities can only be compared for equal or not equal in relational expressions.
`L.EQ.(A+F)`	Logical quantities may never be compared by relational operators.
`E**2 .LT 97.1E1`	There is a missing period immediately after the relational operator.
`.GT.9`	There is a missing arithmetic expression before the relational operator.
`E*2 .EQ. 'ABC'`	A character expression may not be compared to an arithmetic expression.

IBM Extension

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.

End of IBM Extension

Logical Expressions

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

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

Logical Operators

The logical operators are shown in Figure 13. (A and B represent logical constants or variables, or expressions containing relational operators.)

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

Figure 13. Logical Operators

The only valid sequences of two logical operators are:

.AND..NOT.

.OR..NOT.

.EQV..NOT.

.NEQV..NOT.

The sequence .NOT..NOT. is invalid.

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

Examples:

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

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

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

Valid Logical Expressions:

```
(ROOT*A .GT. A) .AND. W
L .AND. .NOT. (I .GT. F)
(E+5.9E2 .GT. 2*E) .OR. L
.NOT. W .AND. .NOT. L
L .AND. .NOT. W .OR. CHAR//'123'.LT.SYMBOL
(A**F .GT. ROOT .AND. .NOT. I .EQ. E)
```

Invalid Logical Expressions:

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

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

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

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

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

Order of Computations in Logical Expressions

In the evaluation of logical expressions, priority of operations involving *arithmetic* operators is as shown in Figure 14. Within a hierarchic level, computation is performed from left to right.

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

Figure 14. Hierarchy of Operations Involving Arithmetic Operators

In the evaluation of logical expressions, priority of operations involving *character* operators is as shown in Figure 15. Within a hierarchic level, computation is performed from left to right.

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

Figure 15. Hierarchy of Operations Involving Character Operators

Example:

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

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

The expression

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

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

1. D**B Call the result W.

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

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

   The real variable A of length 8 is compared to the real variable W, which was extended to a length of 8, yielding a logical result X, whose value is 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, since it is not always necessary to do so to evaluate the expression when A is true.

Use of Parentheses in Logical Expressions

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

Example:

Assume the type of the following variables specified as follows:

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

The expression

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

is evaluated in the following order:

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

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

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

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

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

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

4. .NOT.Z

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

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

   .NOT.(A.OR.B)

   .NOT.A.OR.B

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

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

The lengths of the results of the various logical operations are shown in Figure 16. (The result of logical operations is always logical of length 4.)

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

Figure 16. Type and Length of the Result of Logical Operations

Chapter 5. VS FORTRAN Statements

Source programs consist of a set of statements from which the compiler generates machine instructions and allocates storage for data areas. A VS FORTRAN 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.

VS FORTRAN statements are either executable or nonexecutable.

VS FORTRAN Statement Categories

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

- Assignment statements
- Control statements
- DATA statement

IBM Extension

- Debug statements

End of IBM Extension

- Input/output statements
- PROGRAM statement
- Specification statements
- Subprogram statements

IBM Extension

- VS FORTRAN compiler directive statements

End of IBM Extension

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. Assignment statements are executable.

Control Statements

In the absence of control statements, VS FORTRAN statements are executed sequentially. That is, after one statement has been executed, the statement immediately following it is executed. Control statements alter this normal sequence of execution of statements in the program. They are executable. The following are control statements:

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

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

DATA Statement

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

IBM Extension

Debug Statements

The debug facility is a programming aid that helps locate errors in a VS FORTRAN source program. The debug facility traces the flow of execution within a program, traces the flow of execution between programs, 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.

AT	END DEBUG
DEBUG	TRACE OFF
DISPLAY	TRACE ON

End of IBM Extension

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.

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

BACKSPACE	OPEN
CLOSE	PRINT
ENDFILE	READ
FORMAT	REWIND
INQUIRE	WRITE

IBM Extension

DELETE
REWRITE
WAIT

End of IBM Extension

Note: The description of the VS FORTRAN input and output statements is made from the point of view of a VS FORTRAN 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.)

PROGRAM Statement

The PROGRAM statement names the main program. It can only be used in a main program. It is not required. The PROGRAM statement is nonexecutable.

Specification Statements

The 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.

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

COMMON	EXTERNAL
DIMENSION	IMPLICIT
EQUIVALENCE	INTRINSIC
Explicit type:	PARAMETER
COMPLEX, INTEGER, LOGICAL, REAL, CHARACTER, and DOUBLE PRECISION	SAVE

IBM Extension

NAMELIST

End of IBM Extension

Subprogram Statements

There are three subprogram statements: FUNCTION, SUBROUTINE, and BLOCK DATA. There are also intrinsic function procedures and statement function procedures. The list of intrinsic functions supplied with VS FORTRAN is in Appendix B, "IBM and ANS FORTRAN Features" on page 351.

Function subprograms differ from subroutine subprograms in the way they are invoked and in that function subprograms return a value to the calling program, whereas subroutine subprograms need not return a value.

The function subprogram is a VS FORTRAN subprogram that begins with a FUNCTION statement. It is independently written and is executed whenever its name is appropriately referred to in another program. It is called by coding its name with any necessary parameters. At least one executable statement in the function subprogram must assign a result to the function name; this value is returned to the calling program as the result of the function.

The subroutine subprogram is similar to the function subprogram, except that it begins with a SUBROUTINE statement and does not return an explicit result to the calling program. The rules for naming function and subroutine subprograms are similar. They both require an END statement and they both may contain dummy arguments. Like the function subprogram, the subroutine subprogram can be a set of commonly used computations, but it need not return any results to the calling program. The subroutine subprogram is executed whenever its name is referred to by the CALL statement.

Subprogram statements are nonexecutable.

BLOCK DATA	Statement function
ENTRY	SUBROUTINE
FUNCTION	

IBM Extension

VS FORTRAN Compiler Directive Statements

The EJECT and INCLUDE statements are IBM extensions that direct the compiler to start a new page or to insert one or more source statements into the program. They are not considered part of the VS FORTRAN language.

End of IBM Extension

Order of Statements in a Program Unit

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

1. PROGRAM or subprogram statement, if any.
2. PARAMETER statements and/or IMPLICIT statements, if any.
3. Other specification statements, if any. (Explicit specification statements that initialize variables or arrays must follow other specification statements that contain the same variable or array names.)
4. For the order of data statements, see Figure 17 on page 58.
5. Statement function definitions, if any.
6. Executable statements.
7. END statement.

For the order of DEBUG statements, see "DEBUG Statement" on page 82.

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 name of a constant must precede the PARAMETER statement that defines that particular name of a constant; the PARAMETER statement must precede all other statements containing the names of constants that are defined in the PARAMETER statement.

FORMAT and ENTRY statements may appear anywhere after the PROGRAM or subprogram statement and before the END statement. The ENTRY statement, however, may not appear between a block IF statement and its corresponding END IF statement or within the range of a DO. DATA statements must follow the IMPLICIT statements and specification statements.

IBM Extension

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

End of IBM Extension

The order of statements in BLOCK DATA subprograms is discussed in "BLOCK DATA Statement" on page 69. Figure 17 shows a diagram of the order of statements.

- The vertical lines in the figure delineate varieties of statements that may be interspersed. For example, FORMAT statements may be interspersed with statement function statements and executable statements.

- Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements.

<table>
<tr><td rowspan="5">COMMENT LINES</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, OR BLOCK DATA STATEMENT</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
Statements</td><td rowspan="2">PARAMETER Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="4">END Statement</td></tr>
</table>

Figure 17. Order of Statements and Comment Lines

VS FORTRAN Statement Descriptions

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

Notes:

1 Comments and statement numbers are included because, although they are not actual statements, they are integral parts of VS FORTRAN programs.

2 Most described statements begin at the top of a page.

Arithmetic IF Statement

See "IF Statements" on page 145.

ASSIGN Statement

The ASSIGN statement assigns a number (a **statement number**) to an integer variable. See also "Statement Numbers" on page 236.

Syntax

ASSIGN ***stn*** TO ***i***

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

i
: is the name of an integer variable (not an array element) of length 4 that is assigned the statement number *stn*.

The statement number must be the number of a statement that appears in the same program unit as the ASSIGN statement. The statement number 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 number.

A variable must have been defined with a statement number 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 number may be redefined with the same or a different statement number or an integer value.

If *stn* is the statement number of an executable statement, *i* can be used in an assigned GOTO statement.

If *stn* is the statement number 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 *stn* and cannot be used as such. To use *i* as an integer, it must be assigned an integer value by an assignment or input statement. This assignment can be done directly or through EQUIVALENCE, COMMON, or argument passing.

Valid Example:

These program fragments illustrate the use of the ASSIGN statement to assign the statement numbers of both an executable statement and a FORMAT statement to variables.

```
10 FORMAT (1X, I4)
```

1. Assign statement 30 to integer variable LABEL.

```
ASSIGN 30 TO LABEL
```

2. Assign format statement number 10 to integer variable IFMT.

```
ASSIGN 10 TO IFMT
NUM = 50
```

3. Transfer to statement 30.

```
GOTO LABEL
```

4. Write using the format at statement 10.

```
20 WRITE(5, IFMT) NUM
30 PRINT *, NUM
      END
```

Invalid Example:

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 number.

```
   ASSIGN 10 TO LABEL
10 NUM = 10
```

The following expression is invalid. The results are unpredictable.

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

Assigned GO TO Statement

See "GO TO Statements" on page 142.

Assignment Statements

This VS FORTRAN 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, logical, or character expression.

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

Arithmetic Assignment Statement

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

Figure 18 shows the rules for conversion in arithmetic assignment statements, *a=b*, where the type of *b* is integer or real.

Figure 19 shows the rules for conversion in arithmetic assignment statements, *a=b*, where the type of *b* is complex.

The correspondence between type declarations and data item lengths in bytes is described in Figure 20 on page 107.

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*.

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.

Type of a \ Type of b	INTEGER*2, INTEGER*4, INTEGER	REAL*4, REAL	REAL*8, DOUBLE PRECISION	REAL*16
INTEGER*2, INTEGER*4, INTEGER	Assign	Fix and assign	Fix and assign	Fix and assign
REAL*4, REAL	Float and assign	Assign	Real assign	Real assign
REAL*8, DOUBLE PRECISION	DP float and assign	DP extend and assign	Assign	DP assign
REAL*16	QP float and assign	QP extend and assign	QP extend and assign	Assign
COMPLEX*8, COMPLEX	Float and assign to real part; imaginary part set to 0	Assign to real part; imaginary part set to 0	Real assign real part; imaginary part set to 0	Real assign real part; imaginary part set to 0
COMPLEX*16	DP float and assign to real part; imaginary part set to 0	DP extend and assign to real part; imaginary part set to 0	Assign to real part; imaginary part set to 0	DP assign real part; imaginary part set to 0
COMPLEX*32	QP float and assign to real part; imaginary part set to 0	QP extend and assign to real part; imaginary part set to 0	QP extend and assign real part; imaginary part set to 0	Assign real part; imaginary part set to 0

Figure 18. Conversion Rules for the Arithmetic Assignment Statement a=b, Where Type of b Is Integer or Real

Type of a \ Type of b	COMPLEX*8 / COMPLEX	COMPLEX*16	COMPLEX*32
INTEGER*2 / INTEGER*4 / INTEGER	Fix and assign real part; imaginary part not used	Fix and assign real part; imaginary part not used	Fix and assign real part; imaginary part not used
REAL*4 / REAL	Assign real part; imaginary part not used	Real assign, real part; imaginary part not used	Real assign, real part; imaginary part not used
REAL*8 / DOUBLE PRECISION	DP 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 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	Assign	Real assign real and imaginary parts	Real assign real and imaginary parts
COMPLEX*16	DP extend and assign real and imaginary parts	Assign	DP assign real and imaginary parts
COMPLEX*32	QP extend and assign real and imaginary parts	QP extend and assign real and imaginary parts	Assign

Figure 19. Conversion Rules for the Arithmetic Assignment Statement a=b, Where Type of b Is Complex

Notes to Figures: IBM extensions are shown with inner boxes in the figures. For clarity of presentation, the extensions are not marked in the following definitions. Terms in the figures are defined as follows:

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

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

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

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

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

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

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

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

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

Examples:

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

NAME	TYPE	LENGTH
I, J, K	Integer variables	4, 4 , 2
A, B, C, D	Real variables	4, 4, 8, 8
E	Complex variable	8
F(1),...,F(5)	Real array elements	4
G, H	Logical variables	4, 4
CHAR1	Character variable	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 replaced by as much as can be held in 2 bytes.
A = I	The value of I is converted to a real value, and replaces the value of A.
I = I + 1	The value of I is replaced by the value of I + 1.
E = I**J+D	I is raised to the power J and the result is converted to a real value to which the value of D is added. This result replaces the real part of the complex variable E. The imaginary part of the complex variable is set to zero.
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 zero.
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. is greater than this result, the logical value true replaces the value of G. If 3. 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 for this purpose. See Appendix B, "IBM and ANS FORTRAN Features" on page 351.
F(1) = A	The value of element 1 of array F is replaced by the value of A.
E = F(5)	The real part of the complex constant E is replaced by the value of array element F(5). The imaginary part is set equal to zero.

Statement	Description
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' since CHAR1 is of length 10, and the constant is of length 10.
CHAR1='ABC'	CHAR1 contains the value 'ABCbbbbbbb' since 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' since CHAR1 is of length 10. and the constant is of length 12; the constant is truncated.
CHAR1= 'FGHIJ'//'ABCDE'	CHAR1 contains the value 'FGHIJABCDE', the result of the concatenation operation.

IBM Extension

AT Statement

The AT statement identifies the beginning of a debug packet and indicates the point in the program at which debugging is to begin.

Syntax

AT *stn*

stn

is the number 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 number (*stn*) in the AT statement.

The statement number cannot be specified in another debug packet.

There must be one AT statement for each debug packet; there may be many debug packets for one program or subprogram.

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.

For more on debug packets and for examples of the AT statement, see "DEBUG Statement" on page 82.

End of IBM Extension

BACKSPACE Statement

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

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

Syntax

BACKSPACE *un*

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

UNIT=*un*

un is the reference number of an I/O unit. 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 first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

IOSTAT=*ios*

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

ERR=*stn*

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

Valid BACKSPACE Statements:

```
BACKSPACE un

BACKSPACE (un,ERR=stn)

BACKSPACE (UNIT=un,IOSTAT=ios,ERR=stn)

BACKSPACE (ERR=stn,UNIT=un)

BACKSPACE(UNIT=2*IN+2)

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

Invalid BACKSPACE Statements:

`BACKSPACE UNIT=un`	UNIT= is not allowed without the parentheses.
`BACKSPACE un,ERR=stn`	Parentheses must be specified.
`BACKSPACE (ERR=stn,un)`	UNIT= must be specified.

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 Programming Guide.*) If the unit is not connected, an error is detected.

The external file connected to the unit *un* must exist; otherwise, an error is detected. (The existence of a file can be determined with the INQUIRE statement. *exs* must have the value true. See "INQUIRE Statement" on page 154.)

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.

A BACKSPACE statement for a SYSIN file has no effect.

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. (See "READ Statement—Formatted with Sequential Access" on page 192.)

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. One BACKSPACE may be followed by a WRITE to extend the data set.

IBM Extension

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

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

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

End of IBM Extension

Transfer is made to the statement number specified by the ERR parameter if an error is detected. If IOSTAT=*ios* is specified, a positive integer value is assigned to *ios* when an error is detected. Execution continues with the statement number

specified by the ERR parameter (if present) or with the next statement if the ERR parameter is not specified. If the ERR parameter and the IOSTAT parameter are both omitted, program execution is terminated when an error is detected.

BLOCK DATA Statement

The BLOCK DATA statement names a block of data.

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 the common block name in the executable program. There can only be one unnamed block data subprogram in an

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

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

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

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

Entities not in a named common block must not be initialized and must not appear in a DIMENSION, EQUIVALENCE, or type statement in a block data subprogram.

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 145.

CALL Statement

The CALL statement:

- Transfers control to a subroutine subprogram
- Evaluates actual arguments that are expressions
- Associates actual arguments with dummy arguments

Syntax

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

name

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

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

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

The CALL statement transfers control to the subroutine subprogram and replaces the dummy variables with the values of the actual arguments that appear in the CALL statement.

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 108.)

Valid Examples:

For the following examples, assume that the subroutine definitions below have been defined:

```
SUBROUTINE SUB1
...
END

SUBROUTINE SUB2()
...
END

SUBROUTINE SUB3(A, B, C)
REAL A
REAL B(*)
REAL C(2, 5)
...
END

SUBROUTINE SUB4(LOGL)
LOGICAL LOGL
...
END

SUBROUTINE SUB5(CHAR)
CHARACTER *(*) CHAR
...
END
```

```
SUBROUTINE SUB6(SUBX, X, Y, FUNCX)
EXTERNAL SUBX, FUNCX
Z = FUNCX(X, Y)
CALL SUB7(SUBX)
...
END

SUBROUTINE SUB7(SUBY)
EXTERNAL SUBY
...
CALL SUBY
END

SUBROUTINE SUB8 (A, B, *, *, *)
...
IF(A .LT. 0.0) RETURN 1
IF(A .EQ. 0.0) RETURN 2
RETURN 3
END
```

In the following CALL statement examples that follow, assume that the variable declarations below have been made:

```
DIMENSION W(10), X(10), Z(5)
REAL Y
LOGICAL L
CHARACTER*5 C1, C2
EXTERNAL SUBZ, FUNCA
```

The following CALL statement examples reference the SUBROUTINE declarations above. Some of the examples reference subroutines with an array dimensioned differently than in the calling program, a practice that can cause errors. Variable X in Example 2 below is a case in point. Care must be taken in referencing elements of array X and array C. See "Subscripts" on page 28 for information on array layouts.

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

```
CALL SUB1
CALL SUB1()
CALL SUB2
CALL SUB2()
```

Example with a variable and two array names.

```
CALL SUB3(Y, W, X)
```

Example with an array element and two array names.

```
CALL SUB3(Z(3), X, W)
```

Example with a constant and two array names.

```
CALL SUB3(2.5, W, X)
```

Example with an expression and two array names.

```
CALL SUB3(5*Y, X, W)
```

Example using a logical variable.

```
CALL SUB4(L)
```

Example using a logical constant.

```
CALL SUB4(.FALSE.)
```

Example using a logical expression.

```
CALL SUB4(X(5) .EQ. Y)
```

Example using a character variable.

```
CALL SUB5(C1)
```

Example using a character expression.

```
CALL SUB5(C1 // C2)
```

Example of passing a subroutine name and a function name.

```
CALL SUB6(SUBZ, 1.0, 2.0, FUNCA)
```

Example of passing 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)
```

Invalid Examples:

The following example results indirectly in a call by one subroutine to itself. This is invalid, but cannot be checked by the VS FORTRAN compiler.

```
     CALL SUB6(SUB7, X(5), Y, COS)
```

The following example results in the use of a character variable with implicitly (*) defined length being used in a concatenation operation. This usage is invalid.

```
     SUBROUTINE SUBA(CHAR)
     CHARACTER*(*) CHAR
     CHARACTER*4 C1
     CALL SUBB(CHAR // C1)
     ...
     RETURN
     END
```

Character Type Statement

See "Explicit Type Statement" on page 103.

CLOSE Statement

A CLOSE statement disconnects an external file from an input or output unit.

Syntax

CLOSE ([UNIT=]*un* [,ERR=*stn*] [,STATUS=*sta*] [,IOSTAT=*ios*])

UNIT=*un*

un is the reference to the number of an I/O unit. 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 first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*stn*

is optional. *stn* is the number of an executable statement in the same program unit as the CLOSE statement. If an error is detected, control is transferred to *stn*. If ERR=*stn* is omitted, execution halts when an error is detected.

STATUS=*sta*

is optional. *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.

IOSTAT=*ios*

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

Each of the parameters of the CLOSE statement may appear only once. The unit specifier (*un*) must appear. All value assignments are made according to the rules for assignment statements.

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* may or may not be connected to a file. If the unit is connected, the file may or may not exist.

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, the file is deleted.

If STATUS 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. (The STATUS parameter affects only the internal VS FORTRAN status. The external status is set by the JCL or other system environment and will not be overridden.)

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 **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 provided that the file still exists. (See "OPEN Statement" on page 168.)

When execution ends normally, all units that are connected are closed. 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'

DELETE=STATUS

CLOSE(UNIT=9,STATUS=DELETE)

CLOSE(UNIT=9,STATUS=STATUS)

CLOSE(UNIT=9,STATUS='KEEP')
```

Each of these CLOSE statements should execute the same way and give a status of KEEP.

Comments

Comments provide documentation for a program. They can be entered in either fixed form or free form.

Fixed-Form Input

Fixed-form comments have the following attributes:

- A "C" or an asterisk (*) may appear in column 1, or all blanks may appear in columns 1 to 72.
- A comment may appear anywhere before the END statement.

IBM Extension

Free-Form Input

Free-form comments have the following attributes:

- Any line that does not follow a continued line and that has the quotation mark (") character as its first character is considered a comment.
- A comment line cannot be continued.

Valid Example:

```
Column:  1     7
         ---------------------------------------------------
         "THIS IS A COMMENT
         ...
         10D=010.5
         GOTO 56
         150 A=B+C*(D+E**F--
         G+H-2.*(G+P))
         "THIS IS ANOTHER COMMENT
         ...
         END
```

Invalid Example:

The following example illustrates that a comment cannot follow a line that needs a continuation.

```
Column:  1     7
         ---------------------------------------------------
         "THIS IS A COMMENT
         ...
         10D=010.5
         GOTO 56
         150 A=B+C*(D+E**F--
         "THIS IS NOT A COMMENT IT IS PART OF THE LAST LINE
         G+H-2.*(G+P))
         "THIS IS ANOTHER COMMENT
         ...
         END
```

End of IBM Extension

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 [/[*name*1]/] *list*1[[,] /[*name*n]/ *list*n] ...

name

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

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

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

list

is a list of variable names or array names that are not dummy arguments. If a variable name is also a function name, subroutine name, or entry name, it must not appear in the list. If the list contains an array name, dimensions may also be declared for that array. (See "DIMENSION Statement" on page 87.)

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

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

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

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

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

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

IBM Extension

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

End of IBM Extension

Although the entries in a COMMON statement can contain dimension information, object-time dimensions may never be used.

The length of a blank common can be extended by using an EQUIVALENCE statement, but only by adding beyond the last entry.

A common block resides in a fixed location in storage during the execution of a program. Because of this, all program units of this program refer to data at that location as defined in the COMMON statements in each program unit.

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

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

Main Program / **Subroutine**

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

Blank and Named Common Blocks

Variables and arrays may be placed in separate common blocks by giving distinct common block names (*name*). 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.

IBM Extension

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

End of IBM Extension

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

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

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

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

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

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

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

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

IBM Extension

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

End of IBM Extension

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 103.

Computed GO TO Statement

See "GO TO Statements" on page 142.

CONTINUE Statement

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

Syntax

CONTINUE

CONTINUE

is a statement that may be placed anywhere in the source program (where an executable statement may appear) without affecting the sequence of execution. It may be used as the last statement in the range of a DO loop in order to avoid ending the DO loop with an unconditional or assigned GO TO, block IF, ELSE IF, ELSE, END IF, STOP, RETURN, END, arithmetic IF, another DO statement, or a logical IF statement containing an unconditional or assigned GO TO, or a STOP, RETURN, or arithmetic IF statement.

DATA Statement

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

Syntax

DATA *list*1 /*clist*1/ [[,] *list*2 /*clist*2/] ...

list

is a list of variables, array elements, arrays or substrings, and implied DO lists. (See "Implied DO in a DATA Statement" on page 91.) The comma preceding *list*2...*list*n is optional.

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

clist

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

A DATA initialization statement is not executable. The DATA statement cannot precede a PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA, IMPLICIT, PARAMETER, or an explicit type statement. Otherwise, a DATA statement can appear anywhere in the program.

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

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

IBM Extension

A hexadecimal constant can be used to initialize any type of variable or array element.

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

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

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

End of IBM Extension

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 or array element defined with an initial value may not be in blank common and may not be assigned an initial value more than once. If the variable or array element is in a named common block, it may be initially defined only in a block data subprogram. Because of this constraint, entities that are associated with each other through COMMON or EQUIVALENCE statements are considered to be the same entity.

Valid DATA Statements:

```
LOGICAL L(4)

DIMENSION D(50),F(5),G(9)

CHARACTER*4 C,CC(5)

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/4*.TRUE./, C/'FOUR'/

DATA CC(1)(1:2)/'AB'/,CC(1)(3:4)/'CD'/
```

IBM Extension

```
DATA CC(2)/ZC5C6C7C8/,I/ZF8/,R/Z00/
```

End of IBM Extension

IBM Extension

DEBUG Statement

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

Syntax

DEBUG *option*1[,*option*2...]

An *option* may be any of the following:

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

SUBCHK (*a*1, *a*2,..., *a*n)
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 number within this program. Even when this option is used, a TRACE ON statement must appear in the first debug packet in which tracing is desired.

INIT (*i*1, *i*2,..., *i*n)
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 message RETURN is to be displayed whenever execution of the subprogram is completed.

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

All debugging statements must precede the first statement of the program being debugged.

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

1. DEBUG statement

2. Debug packets

3. END DEBUG statement

4. First of the source program statements of a program unit to be debugged

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

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

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

TRACE ON
TRACE OFF
DISPLAY

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

Considerations When Using DEBUG

The following precautions must be taken when setting up a debug packet:

- Any DO loops or block IF, ELSE IF, or ELSE statements initiated within a debug packet must be wholly contained within that packet.

- Statement numbers within a debug packet must be unique. They must be different from statement numbers 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 number defined in a debug packet; however, control may be returned to any point in the program being debugged from a packet. In addition, no debug packet may refer to a label defined in another debug packet. A debug packet may contain a RETURN, STOP, or CALL statement.

- The SUBCHK function of DEBUG does proper subscript checking of an array if, and only if, that array is a single-dimensioned array with a lower bound of 1. If the lower bound is not 1 and an error is detected, the message will give the index to the element as if it had a lower bound of 1. If 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
```

This example checks all arrays for valid subscripts.

Example 2:

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

The values of A, B, and C are to be examined as they were at the completion of the arithmetic operation in statement 10. Therefore, the statement number 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
```

When statement 10 is encountered, tracing begins, as specified by the TRACE ON statement in the first debug packet. When statement 25 is encountered, tracing stops, as specified by the TRACE OFF statement in the second debug packet. When 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.

End of IBM Extension

IBM Extension

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=*stn*])

UNIT=*un*

un is the reference number of an I/O unit. 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 first in the statement. The other parameters can appear in any order. If UNIT= is specified, all the parameters can appear in any order.

IOSTAT=*ios*

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

ERR=*stn*

stn is the number of an executable statement in the same program unit as the DELETE statement. If an error is detected, control is transferred to *stn*.

Valid DELETE Statement:

```
DELETE (15)
```

End of IBM Extension

DIMENSION Statement

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

Syntax

DIMENSION *a*1(*dim*1) [, *a*2(*dim*2)] ...

a

is an array name.

dim

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

```
   e1:e2
or
   e2
```

where:

e1

is the lower dimension bound. It is optional. If *e*1 (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.

(See "Size and Type Declaration of an Array" on page 30 for rules about dimension bounds.)

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

Valid DIMENSION Statements:

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

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

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

DIMENSION G(I:J,M:N)

DIMENSION ARRAY (M*N:I*J)

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

IBM Extension

DISPLAY Statement

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

Syntax

DISPLAY *list*

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

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

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

NAMELIST /*name*/*list*

WRITE (*un*, *name*)

where *name* is the same in both statements.

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

For examples and explanations of the DISPLAY statement, see "DEBUG Statement" on page 82.

End of IBM Extension

DO Statement

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

Syntax

	End of Range		DO Variable		Initial Value	Test Value	Increment
DO	*stn*	[,]	*i*	=	*e*1,	*e*2	[,*e*3]

stn
: is the number of an executable statement, in the same program unit as the DO statement, that denotes the end of the DO loop. The statement at *stn* cannot be an unconditional or assigned GOTO, block IF, ELSEIF, ENDIF, STOP, RETURN, END, arithmetic IF, another DO statement, or a logical IF statement containing an unconditional or assigned GOTO, STOP, RETURN, or arithmetic IF statement.

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

e1*, *e2*, and *e3
: are integer, real, or double precision arithmetic expressions that define the DO-loop iteration. *e*3 is optional and cannot have a value of zero; if it is

omitted, its value is assumed to be 1, and the preceding comma must be omitted. The expressions ***e*1**, ***e*2**, and ***e*3** are evaluated, and the control parameters ***m*1**, ***m*2**, and ***m*3**, respectively, are determined from them. The expressions ***m*1**, ***m*2**, and ***m*3** are converted to the type of the DO variable, where the data types are not consistent.

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

*m*1 is less than or equal to *m*2, and *m*3 is greater than zero
or
*m*1 is greater than or equal to *m*2, and *m*3 is less than zero.

If one of the above relationships is true, the first time the statements in the range of the DO are executed, *i* is initialized to the value of *m*1; on each succeeding iteration, *i* is increased by the value of *m*3. The number of iterations that can be executed, called the iteration count, is the value of:

MAX (INT((*m*2 - *m*1 + *m*3) / *m*3), 0).

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

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

The DO variable may not be redefined within the range of the DO loop. However, any of the variables in the expressions ***e*1**, ***e*2**, and ***e*3** may be modified by the statements in the DO loop without changing the iteration count as established for the DO statement. To exit the DO loop before all iterations are completed, a transfer instruction (GOTO, computed GOTO, assigned GOTO, CALL with return values, arithmetic IF) must be executed, which transfers out of the range of the DO.

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

Valid Examples:

The following program fragment illustrates the use of real expressions when defining the DO control parameters and the DO variable.

```
      XEND = 10.5
      XINCR = .5
      J = 0
      DO 10 X = 1.0, XEND, XINCR
   10 J = J + 1
```

The iteration count for the above example is 20; that is,

Iteration Count = MAX(INT((10.5 - 1.0 + .5)/.5), 0) = 20

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

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

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

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

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

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

Results from the nested DO example:

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

Implied DO in a DATA Statement

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

Syntax

(*dlist*, *i* = *m*1, *m*2 [, *m*3])

where:

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

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

***m*1, *m*2, and *m*3**
: are each integer constants or names of integer constants, or expressions containing only integer constants or names of integer constants. An expression may contain implied DO variables of other surrounding implied

DO lists that have this implied DO list within their ranges (dlist). *m*3 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 *m*1, *m*2, and *m*3 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

Invalid Implied DO Statement:

```
DATA (K(I),I=1,3),(L(I),I=1,3),(M(I),I=1,2)/8*1/
```

The two DO lists, (K(I),I=1,3) and (L(I),I=1,3), cannot share the same DO variable (I) if they also use the same list of constants (/8*1/).

Implied DO in an Input/Output Statement

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

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

where:

dlist
is an input/output list.

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

***m*1, *m*2,** and ***m*3**
: are integer, real, or double precision arithmetic expressions. The values of the expressions *m*1, *m*2, and *m*3 are converted to the type of the DO variable *i*, if necessary. *m*3 is optional and cannot have a value of zero; if it is omitted, its value is assumed to be 1, and the preceding comma must be omitted.

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

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

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

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

Or the statement:

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

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

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

Implied DOs can be nested, if required. For example, to read an element into array B after values are read into each row of a 10x20 array A, the following input statement would be written:

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

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

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

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

Double Precision Type Statement

See "Explicit Type Statement" on page 103.

IBM Extension

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
```

End of IBM Extension

ELSE Statement

See "IF Statements" on page 145.

ELSE IF Statement

See "IF Statements" on page 145.

END Statement

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

Syntax

```
END
```

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

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

- Entities specified in SAVE statements. (See "SAVE Statement" on page 232.)
- Entities in a blank common block.
- Initially defined entities that have neither been redefined nor become undefined.

- Entities in named common blocks that appear in the subprogram and appear in at least one other program unit that is referring, either directly or indirectly, to that subprogram. The entities in a named common block may become undefined by execution of a RETURN or END statement in another program unit.

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

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

END Statement in a Function Subprogram

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

A subprogram must not be referred to twice during the execution of an executable program without the intervening execution of a RETURN or END statement in that subprogram.

END Statement in a Subroutine Subprogram

All subroutine subprograms must end with END statements. They may also contain RETURN statements. An END statement specifies the physical end of the subprogram. If the END statement is reached during execution of the subroutine subprogram, it is executed as a RETURN statement.

IBM Extension

END DEBUG Statement

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

Syntax

END DEBUG

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

See "DEBUG Statement" on page 82.

End of IBM Extension

ENDFILE Statement

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

Syntax

ENDFILE *un*

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

UNIT=*un*

un is the reference to the number of an I/O unit. 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 first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*stn*

is optional. *stn* is the number of an executable statement in the same program unit as the ENDFILE statement. If ERR=*stn* is omitted, execution halts when an error is detected.

IOSTAT=*ios*

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

Valid ENDFILE Statements:

```
ENDFILE un

ENDFILE (un,ERR=stn)

ENDFILE (UNIT=un,ERR=stn)

ENDFILE (ERR=stn,UNIT=un)
```

Invalid ENDFILE Statements:

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

ENDFILE un,ERR=stn        Parentheses must be specified.

ENDFILE (ERR=stn,un)      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 Programming Guide* for an example.) If the unit is not connected, an error is detected.

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.

IBM Extension

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.

Multiple file data sets are permitted in VS FORTRAN. Therefore, after execution of an ENDFILE, additional data may be transferred to the subsequent files.

End of IBM Extension

Transfer is made to the statement specified by the ERR= if an error is detected. If 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 parameter, if present, or with the next statement if ERR is not specified. If the ERR parameter and the IOSTAT parameter are both omitted, program execution is terminated when an error is detected.

END IF Statement

See "IF Statements" on page 145.

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* [([*arg*1 [, *arg*2] ...])]

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 56. 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 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 30.

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 Statement Examples:

To illustrate the use of the ENTRY within a subroutine subprogram, the following subprogram is defined:

```
      SUBROUTINE SAMPLE(A,I,C)
      X = A**I
      GO TO 10
      ENTRY ALIAS(B,C)
      X = B
   10 C = SQRT(X)
      RETURN
      END
```

The subprogram invocation

```
CALL SAMPLE(X,J,Z)
```

evaluates the expression SQRT(X**J) and returns the value in Z.

The subprogram invocation

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

evaluates the expression SQRT(Y) and returns the value in W.

Actual Arguments in an ENTRY Statement

Entry into a function subprogram associates actual arguments with the dummy arguments of the referenced ENTRY statement. Thus, all appearances of these arguments in the subprogram become associated with actual arguments.

See "Actual Arguments in a Subroutine Subprogram" on page 239 and "Actual Arguments in a Function Subprogram" on page 140.

Dummy Arguments in an ENTRY Statement

The dummy arguments in the ENTRY statement need not agree in order, type, or number with the dummy arguments in the SUBROUTINE or FUNCTION statement or any other ENTRY statement in the same subprogram. However, the actual arguments for each CALL or function reference must agree in order, type, and number with the dummy arguments in the SUBROUTINE, FUNCTION, or ENTRY statement to which it refers.

Any dummy argument of an ENTRY statement must not be in an executable statement preceding the ENTRY statement unless it has already appeared as a dummy argument in an ENTRY, SUBROUTINE, or FUNCTION statement prior to the executable 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 specifier.

A dummy argument must not appear in the expression of a statement function definition 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 used in an executable statement is allowed only if that dummy argument appears in the argument list of the FUNCTION, SUBROUTINE, or ENTRY statement by which the subprogram was entered.

See "Dummy Arguments in a Subroutine Subprogram" on page 239 and "Dummy Arguments in a Function Subprogram" on page 140.

EQUIVALENCE Statement

The EQUIVALENCE statement permits the sharing of data storage within a single program unit.

Syntax

EQUIVALENCE (*list*1) [, (*list*2)] ...

list

is a list of variable, array, array element, or character substring names. Names of dummy arguments of an external procedure in a subprogram must not appear in the list. Each pair of parentheses must contain at least two names.

The number of subscript quantities of array elements must be equal to the number of dimensions of the array. If an array name is used without a subscript in the EQUIVALENCE statement, it is interpreted as a reference to the first element of the array.

An array element refers to a position in the array in the same manner as it does in an assignment statement (that is, the array subscript specifies a position relative to the first element of each dimension of the array).

The subscripts and substring information may be integer expressions containing only integer constants, or names of integer constants. They must not contain variables, array elements, or function references.

All the named data within a single set of parentheses shares the same storage location. When the logic of the program permits it, the EQUIVALENCE statement can reduce the number of bytes used by sharing two or more variables of the same type or different noncharacter types.

IBM Extension

Both character and noncharacter data types are allowed in an EQUIVALENCE relationship.

End of IBM Extension

The length of the equivalenced entities can be different. Equivalence between variables implies storage sharing.

Mathematical equivalence of variables or array elements is implied only when they are of the same noncharacter type, when they share exactly the same storage, and when the value assigned to the storage is of that type.

Because arrays are stored in a predetermined order, equivalencing two elements of two different arrays implicitly equivalences other elements of the two arrays. The EQUIVALENCE statement must not contradict itself or any previously established equivalences.

EQUIVALENCE

Two variables in one common block or in two different common blocks cannot be made equivalent. However, a variable in a program unit can be made equivalent to a variable in a common block. If the variable that is equivalenced to a variable in the common block is an element of an array, the implicit equivalencing of the rest of the elements of the array can extend the size of the common block. The size of the common block cannot be extended so that elements are added ahead of the beginning of the established common block.

For the following examples of the EQUIVALENCE statement, assume these explicit type declarations:

```
COMMON /COM1/ B(50,50), E(50,50)
INTEGER*4 A(10)
REAL*8 C(50), D(10,10,2), F
CHARACTER*4 C1(10), C2(10)
CHARACTER C3
```

Valid Examples

1. A locally defined variable sharing named common storage.

```
EQUIVALENCE (A(1), E(1,1))
```

2. Equivalence a portion of a multidimensioned 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. 20 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: ))
```

IBM Extension

Character variables may be equivalenced to noncharacter items.

A character array is equivalenced to the second half of an integer array.

```
EQUIVALENCE (C1(1), A(6))
```

End of IBM Extension

Invalid Example

Two variables may not be equivalenced when both are in common.

```
EQUIVALENCE (B(1,1), E(1))
```

Explicit Type Statement

The explicit type statement:

- Specifies the type and length of variables, arrays, and user-supplied functions.
- Specifies the dimensions of an array.

IBM Extension

- Assigns initial data values for variables and arrays.

End of IBM Extension

The explicit type statement overrides the IMPLICIT statement, which, in turn, overrides the predefined convention for specifying type.

Syntax

type *name*1 [, *name*2] ...

type

is complex, integer, logical, real, double precision, or character[**len*[,]]

where:

len

specifies the length (number of characters between 1 and 32767). It is optional.

Note: The CHARLEN compiler option may be specified to set 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.

The length *len* can be expressed as:

- An unsigned, nonzero, integer constant.

- An expression with a positive value that contains integer constants, names of integer constants enclosed in parentheses, or an asterisk enclosed in parentheses.

The length **len* immediately following the word character is used as the length specification of any name in the statement that has no length specification attached to it. To override a length for a particular name, see the alternative forms of *name* below. If **len* is not specified, it is assumed to be 1.

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

IBM Extension

type

is complex[**len*1], integer[**len*1], logical[**len*1], or real[**len*1]

where:

***len*1**

is optional and ***len*1** represents one of the permissible length specifications for its associated type as described in Figure 4 on page 26.

End of IBM Extension

name

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

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

or

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

where:

a

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

dim

is optional. *dim* may only be specified for arrays. It is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array in the form:

```
e1:e2
```

or

```
e2
```

where:

e1

is the lower dimension bound. It is optional. If *e*1 (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.

(See "Size and Type Declaration of an Array" on page 30 for rules about dimension bounds.)

If a specific intrinsic function name appears in an explicit specification statement that causes a conflict with the type specified for this function in Appendix B, "IBM and ANS FORTRAN Features" on page 351, 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, then 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 137).

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 (*).

IBM Extension

name

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

a[**len*3][(*dim*)]

or

a[**len*3][(*dim*)] [/*i*1,*i*2,*i*3,...,*i*n/]

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[**len*1], character[**len*], integer[**len*1], logical[**len*1], or real[**len*1]

dim

is optional. *dim* may only be specified for arrays. It is composed of one through seven dimension bounds, separated by commas, that represent the limits for each subscript of the array. See the description of *dim* above.

***i*1,*i*2,*i*3,...,*i*n**

are optional and represent initial data values.

Dummy arguments and names of constants, functions and statement functions may not be assigned initial values.

Initial data values may be assigned for any items of type double precision.

Initial data values may be assigned to variables or arrays that are not dummy arguments or in blank common, by use of *i*n, where *i*n is a constant or list of constants separated by commas. Each *i*n provides initialization only for the immediately preceding variable or array. Lists of constants are used only to assign initial values to array elements. The data must be of the same type as the variable or array, except that hexadecimal data may also be used.

Note: If hexadecimal data is used, the hexadecimal constant form must be followed. (See "Hexadecimal Constants" on page 23.)

Successive occurrences of the same constant can be represented by the form i*constant, as in the DATA statement. If initial data values are assigned to an array in an explicit specification statement, the dimension information for the array must be in the explicit specification statement or in a preceding DIMENSION or COMMON statement.

End of IBM Extension

The following table lists all the possible explicit type statements, and the resulting type and length of the data item.

Type Statement	**Resulting Type**	**Length (Bytes)**
CHARACTER	CHARACTER	1
CHARACTER*n	CHARACTER	n (where $1 \leq n \leq x$)[1]
COMPLEX	COMPLEX	8
COMPLEX*8	COMPLEX	8
COMPLEX*16	COMPLEX	16
COMPLEX*32	COMPLEX	32
DOUBLE PRECISION	REAL	8
INTEGER	INTEGER	4
INTEGER*2	INTEGER	2
INTEGER*4	INTEGER	4
LOGICAL	LOGICAL	4
LOGICAL*1	LOGICAL	1
LOGICAL*4	LOGICAL	4
REAL	REAL	4
REAL*4	REAL	4
REAL*8	REAL	8
REAL*16	REAL	16

[1]If the CHARLEN compiler option is not specified, x=500. If CHARLEN is specified, x=CHARLEN, where x is less than 32,768. For more information about the CHARLEN option, see *VS FORTRAN Programming Guide*.

Valid Explicit Type Statements:

```
CHARACTER*8ORANGES

DATA ORANGES/'ORANGES  '/

CHARACTER*8ORANGES/'ORANGES  '/

SUBROUTINE SUB(DUM)
CHARACTER *(*) DUM
```

IBM Extension

```
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)
```

End of IBM Extension

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 *name*1 [, *name*2] ...

name
: is a name of a user-supplied subprogram (function or subroutine) that is passed as an argument to another subprogram.

EXTERNAL is a specification statement and must precede DATA statement, statement function definitions, and all executable statements.

Statement function names cannot appear in EXTERNAL statements. If the name of a VS FORTRAN-supplied function (that is, intrinsic function) is used in an EXTERNAL statement, the function is no longer recognized as being an intrinsic function when it appears as a function reference. Instead, it is assumed that the function is supplied by the user.

The same name may not appear in both an EXTERNAL and an INTRINSIC statement.

The name of any subprogram that is passed as an argument to another subprogram must appear in an EXTERNAL or INTRINSIC statement in the calling program.

Valid EXTERNAL Statement:

```
EXTERNAL TREES
```

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 (*f*1 [, *f*2 [,..., *f*n]])

*f*1, *f*2,..., *f*n are format codes.

Code	Format	Description
I	*a*I*w*	Integer data fields
I	*a*I*w*.*m*	Integer data fields
D	*a*D*w*.*d*	Double precision data fields
E	*a*E*w*.*d*	Real data fields
E	*a*E*w*.*d*E*e*	Real data fields
F	*a*F*w*.*d*	Real data fields
G	*a*G*w*.*d*	Real data fields
G	*a*G*w*.*d*E*e*	Real data fields
P	*n*P	Scale factor
L	*a*L*w*	Logical data fields
A	*a*A	Character data fields
A	*a*A*w*	Character data fields
	'character constant'	Literal data (character constant)
H	*w*H	Literal data (Hollerith constant)
X	*w*X	Input: Skip a field Output: Fill with blanks
T	T*r*	Transfer of data starts in current position
TL	TL*r*	Transfer of data starts *r* characters to the left of current position
TR	TR*r*	Transfer of data starts *r* characters to the right of current position
group	*a*(...)	Group format specification
S	S	Display of optional plus sign is restored
SP	SP	Plus sign is produced in output
SS	SS	Plus sign is not produced in output
BN	BN	Blanks are ignored in input
BZ	BZ	Blanks are treated as zeros in input
slash	/	Data transfer on the current record is ended
colon	:	Format control is terminated if there are no more items in the input/output list

IBM Extension

Code	Format	Description
E	*a*E*w*.*d*D*e*	Real data fields
G	*a*G*w*.*d*	Integer or logical data fields
G	*a*G*w*.*d*E*e*	Integer or logical data fields
Q	*a*Q*w*.*d*	Extended precision data fields
Z	*a*Z*w*	Hexadecimal data fields

End of IBM Extension

a
is an optional repeat count—an unsigned, nonzero, integer constant used to denote the number of times the format code or group is to be used. The range of *a* is 1 to 255. If *a* is omitted, the code or group is used only once.

w
is an unsigned, nonzero, integer constant that specifies the width of the field.

m
is an unsigned integer constant that specifies the number of digits to be printed.

d
is an unsigned integer constant that specifies the number of digits to the right of the decimal point.

e
is an unsigned, nonzero, integer constant that specifies the number of digits in the exponent field.

n
is an (optionally) signed integer constant that specifies a scale factor to be applied.

r
is an unsigned, nonzero, integer constant that specifies a character position in a record.

(...)
is a group format specification. Within the parentheses are format codes or additional levels of groups, separated by commas, slashes, or colons. Commas are optional before or after a slash and before or after a colon, if the slash or colon is not part of a character constant.

The FORMAT statement is used with READ and WRITE statements for internal and external files. The external files must be connected for SEQUENTIAL or DIRECT access. In the FORMAT statement, the data fields are described with format codes, and the order in which these format codes are specified determines

the structure of the FORTRAN records. The I/O list gives the names of the data items that make up the record. The length of the list, in conjunction with the FORMAT statement, specifies the length of the record. (See "Forms of a FORMAT Statement" on page 114.)

The format codes delimited by left and right parentheses may appear as a character constant in the format specification of the READ or WRITE statement, instead of in a separate FORMAT statement. For example,

```
READ (UNIT=5,FMT='(I3,F5.2,E10.3,G10.3)')N,A,B,C

READ (5,'(I3,F5.2,E10.3,G10.3)')N,A,B,C
```

Throughout this section, the examples show punched card input and printed line output. However, the concepts apply to all input/output media. the examples, the character b represents a blank.

General Rules for Data Conversion

The following is a list of general rules for using the FORMAT statement or a format in a READ or WRITE statement.

- FORMAT statements are not executed; their function is to supply information to the object program. They may be placed anywhere in a program unit other than in a block data subprogram, subject to the rules for the placement of the PROGRAM, FUNCTION, SUBROUTINE, and END statements.

- Complex data in records requires two successive D, E, G, or F format codes.

IBM Extension

VS FORTRAN also accepts the Q format code for complex data.

End of IBM Extension

  The two codes may be different and the format codes T, TL, TR, X, /, :, S, SP, SS, P, BN, BZ, H, or a character constant may appear between the two codes.

- When defining a VS FORTRAN record by a FORMAT, it is important to consider the maximum size record allowed on the input/output medium. For example, if a VS FORTRAN 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. For input, the FORMAT should not define a VS FORTRAN record longer than the actual input record.

- When records are to be printed, the first character of each record functions as a carrier control character. The control character determines the vertical spacing of the printed record and is not considered as part of a data item, as follows:

Control Character	*Vertical Spacing Before Printing*
blank	Advance one line.
0	Advance two lines.
1	Advance to first print position on next page.
+	No advance (overstrike).

The control character is commonly specified in a FORMAT statement, using either of two forms of character constant data, 'x' or 1Hx, where x is one of the characters shown above. The characters and spacing shown are those defined for VS FORTRAN print records, and the result of using other characters in the control position is indeterminate (except that the control position is always discarded). If the print record contains no characters, then spacing is advanced by one, and a blank line is printed.

IBM Extension

If records are to be displayed at a terminal, control characters are also employed, and characters blank and zero (only) produce the spacing shown above when used in the control position.

End of IBM Extension

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 127).

    To produce a blank record on output, an empty format specification of the form FORMAT ( ) may be used.

- To illustrate the nesting of group format specifications, the following statements are both valid:

```
FORMAT (...,a(...,a(...),...,a(...),...))
```

or

```
FORMAT (...,a(...,a(...,a(...),...),...),...)
```

where *a* is $1 \leq a < 256$.

- To illustrate the use of nesting in an implied DO and the corresponding FORMAT specifications:

```
      PROGRAM FMT1
      DIMENSION IRR(3,4), IRI(3,4)
      DO 10 I = 1, 3
      DO 10 J = 1, 4
      IRR(I,J) = 1000 + (I * 100) + J
      IRI(I,J) = 2000 + (I * 100) + J
 10   CONTINUE
      PRINT  20, (I, (IRR(I,J), IRI(I,J), J = 1, 4),
     1  I = 1, 3)
 20   FORMAT (3(1X, 'ROW', I3, 4( I5, 1X, I4, 3X) / ))
      STOP
      END
```

  Results of program FMT1:

```
ROW  1 1101 2101    1102 2102    1103 2103    1104 2104
ROW  2 1201 2201    1202 2202    1203 2203    1204 2204
ROW  3 1301 2301    1302 2302    1303 2303    1304 2304
```

- Names of constants must not be a part of a format specification (see "PARAMETER Statement" on page 173).

- 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 may be omitted. A field of all blanks is considered to be zero.

    With F, E, G, and D format codes, a decimal point appearing in the input field overrides the portion of a format code that specifies the decimal point location. The input field may have more digits than VS FORTRAN uses to approximate the value.

  - **Output**: The representation of a positive or zero internal value in the field may be prefixed with a plus, as controlled by the S, SP, and SS format codes. The representation of a negative internal value in the field is prefixed with a minus. A negative zero is not produced.

    The representation is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks are inserted in the field.

    If the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w*.*d*E*e* or G*w*.*a*E*e* format codes, the entire field of width *w* is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, asterisks are not produced. When an SP format code is in effect, a plus is not optional.

IBM Extension

With VS FORTRAN, 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 uses to approximate the value.

- **Output**: If the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w*.*d*D*e* or Q*w*.*d* format codes, the entire field of width *w* is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, asterisks are not produced. When an SP format code is in effect, a plus is not optional.

End of IBM Extension

Forms of a FORMAT Statement

All the format codes in a FORMAT statement are enclosed in parentheses. Within these parentheses, the format codes are delimited by commas. The comma used to separate list items may be omitted as follows:

- Between a P edit descriptor and an immediately following F, E, D, or G format code

- Before or after a slash format code

- Before or after a colon format code

Execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information provided jointly by the I/O list, if one exists, and the format specification. If there is an I/O list, there must be at least one I, D, E, F, A, G, or L format code in the format specification.

IBM Extension

The Q and Z format codes may also appear in the format specification.

End of IBM Extension

There is no I/O list item corresponding to the format codes: T, TL, TR, X, H, character constants enclosed in apostrophes, S, SP, SS, BN, BZ, P, the slash (/), or the colon (:). These communicate information directly to the record.

Whenever an I, D, E, F, A, G, or L format code is encountered, format control determines whether there is a corresponding element in the I/O list.

IBM Extension

With VS FORTRAN, the list of format codes includes Q and Z.

Whenever a Q or Z code is encountered, format control determines whether there is a corresponding element in the I/O list.

The comma may be omitted between a P format code and an immediately following Q format code.

End of IBM Extension

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, or L format code or the final right parenthesis of the format specification is encountered.

IBM Extension

The question is also asked when a Q or Z format code is encountered.

End of IBM Extension

Before this is done, T, TL, TR, X, and H codes, character constants enclosed in apostrophes, colons, and slashes are processed. If there are fewer elements in the I/O list than there are format codes, the remaining format codes are ignored.

I Format Code

The I format code edits integer data. For example, if a READ statement refers to a FORMAT statement containing I format codes, the input data is stored in internal storage in integer format. The magnitude of the data to be transmitted must not exceed the maximum magnitude of an integer constant.

Input: Leading blanks in a field of the input line are interpreted as zeros. Embedded and trailing blanks are treated as indicated in the general rules for numeric fields described under "General Rules for Data Conversion" on page 111. If the form I*w.m* is used, the value of *m* has no effect.

Output: The output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise. If the number of significant digits and sign required to represent the quantity in the datum is less than *w*, the unused leftmost print positions are filled with blanks. If it is greater than *w*, asterisks are printed instead of the number. If the form I*w.m* is used, the output is the same as the I*w* form, except that the unsigned integer constant consists of at least *m* digits and, if necessary, has leading zeros. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

F Format Code

The F*w.d* format code edits real data. It indicates that the field occupies *w* positions, the fractional part of which consists of *d* digits.

Input: The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented.

The input field may have more digits than VS FORTRAN uses to approximate the value of the datum. The basic form may be followed by an exponent of one of the following forms:

- Signed integer constant.
- E followed by zero or more blanks, followed by an optionally signed integer constant.
- D followed by zero or more blanks, followed by an optionally signed integer constant.

IBM Extension

- Q followed by zero or more blanks, followed by an optionally signed integer constant.

End of IBM Extension

An exponent containing a D is processed identically to an exponent containing an E.

IBM Extension

An exponent containing a Q is processed identically to an exponent containing an E.

End of IBM Extension

Output: The output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise. This is followed by a string of digits that contains a decimal point, representing the magnitude of the internal value, as modified by the established scale factor and rounded to *d* fractional digits. Leading zeros are not provided, except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero also appears if there would otherwise be no digits in the output field.

D, E, and Q Format Codes

The D*w.d*, E*w.d*, E*w.d*E*e* format codes edit real, complex, or double precision data.

IBM Extension

The E*w.d*D*e* and Q*w.d* format codes edit **extended precision data** in addition to real, complex, and double precision data.

End of IBM Extension

The external field occupies *w* positions, the fractional part of which consists of *d* digits (unless a scale factor greater than 1 is in effect). The exponent part consists of *e* digits. (The *e* has no effect on input.)

Input: The input field may have more digits than VS FORTRAN uses to approximate the value of the datum.

Input datum must be a number, which, optionally, may have a D or E exponent, or which may be omitted from the exponent if the exponent is signed.

IBM Extension

It may also have a Q exponent.

End of IBM Extension

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 111.

The input datum may have an exponent of any form. The input datum is converted to the length of the variable as specified in the I/O list. The *e* of the exponent in the format code has no effect on input.

Output: For data written under a D or E format code, unless a P-scale factor is in effect, output consists of an optional sign (required for negative values), an optional zero digit, a decimal point, the number of significant digits specified by *d*, and a D or E exponent requiring four positions.

If the P-scale factor is negative, output consists of an optional sign (required for negative values), an optional zero digit, a decimal point, $|P|$ leading zeros, $|d+P|$ significant digits, and a D or E exponent requiring four positions. (***P*** is the value of the P-scale factor.)

If the P-scale factor is positive, output consists of an optional sign (required for negative values), ***P*** decimal digits, a decimal point, $d-P+1$ fractional digits, and a D or E exponent requiring four positions. (***P*** is the value of the P-scale factor.)

IBM Extension

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.

End of IBM Extension

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. A zero is placed to the left of the decimal point, if the output field consists only of a fractional segment, and if additional space is available. If the entire value is zero, a zero is printed before the decimal point.

G Format Code

The G format code is a generalized code used to **transmit real data** according to the type specification of the corresponding variable in the I/O list. The G*w.d* and G*w.d*E*e* edit descriptors indicate that the external field occupies *w* positions. Unless a scale factor greater than one is in effect, the fractional part of *w* consists of *d* digits. The exponent part consists of *e* digits.

Input: The form of the input field is the same as for the F format code.

Output: The method of representation in the output field depends on the magnitude of the data being edited.

For example, letting N be the magnitude of the internal data,

```
if N < 0.1   or   N ≥ 10**d
```

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

- G*w.d* output editing is the same as *k*PE*w.d* output editing.
- G*w.d*E*e* output editing is the same as *k*PE*w.d*E*e* output editing.

If N is greater than or equal to 0.1 and less than 10***d*, the scale factor has no effect, and the value of N determines the editing as follows:

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

b means blank.

n means:

- 4 for G*w.d*
- *e*+2 for G*w.d*E*e*

The scale factor has no effect unless the magnitude of the data to be edited is outside the range that permits effective use of F editing.

IBM Extension

The letter Q is used for the exponent of extended precision data.

The G format code may be used to transmit integer or logical data according to the type specification of the corresponding variable in the I/O list.

If the variable in the I/O list is integer or logical, the *d* portion of the format code, specifying the number of significant digits, can be omitted; if it is given, it is ignored.

End of IBM Extension

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, and G format codes until another scale factor is encountered; then that scale factor is established.

IBM Extension

It also applies to all subsequently interpreted Q format codes.

End of IBM Extension

Reversion of format control does not affect the established scale factor. A repetition code can precede these format codes. For example, 2P,3F7.4 is valid. (A comma must be placed after the P format code—for example, 2P,3F7.4—when a repeat count is specified.) A scale factor of zero may be specified.

Input: If an exponent is in the data field, the scale factor has no effect. If no exponent is in the field, the externally represented number equals the internally represented number multiplied by 10***n* for the external representation.

For example, if the input data is in the form

```
xx.xxxx
```

and is to be used internally in the form

```
  .xxxxxx
```

then the format code used to effect this change is

```
2PF7.4
```

which may also be written 2P,F7.4.

Similarly, if the input data is in the form

```
  xx.xxxx
```

and is to be used internally in the form

```
xxxx.xx
```

then the format code used to effect this change is

```
-2PF7.4
```

which also may be written -2P,F7.4.

Output: With an F format code, the internally represented number reduced by 10**n is produced.

For example, if the number has the internal form

```
 .xxxxxx
```

and is to be written in the form

```
xx.xxxx
```

the format code used to effect this change is

```
2PF7.4
```

which also may be written 2P,F7.4.

On output with E and D 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*.

IBM Extension

On output with Q format code, the value of the internally represented number is not changed.

End of IBM Extension

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 G*w.d*.

If $.1 \leq m < 10**d$ and the F format code is used, there is **no** difference between G format code **with** a scale factor and G format code **without** a scale factor.

However, if $m \geq 10**d$ or < 0.1, the scale factor moves the decimal point to the right or left.

The following example illustrates the difference between G format code with and without a scale factor:

If A is initially set to 100 and multiplied by 10 each time, and:

```
76  FORMAT ('  ',G13.5,1PG13.5,2PG13.5)
    WRITE  (6,76) A,A,A
```

the result is:

No Scale Factor	Scale Factor = 1	Scale Factor = 2
100.00	100.00	100.00
1000.0	1000.0	1000.0
10000.	10000.	10000.
0.10000E+06	1.00000E+05	10.0000E+04
0.10000E+07	1.00000E+06	10.0000E+05

IBM Extension

Z Format Code

The Z format code **transmits hexadecimal data.**

Input: Scanning of the input field proceeds from right to left. Leading, embedded, and trailing blanks in the field are treated as zeros. One byte in internal storage contains two hexadecimal digits; thus, if an input field contains an odd number of digits, the number is padded on the left with a hexadecimal zero when it is stored. If the storage area is too small for the input data, the data is truncated and high-order digits are lost.

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.

End of IBM Extension

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 114 explains how to define more than one record format in a FORMAT statement.

IBM Extension

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 and 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.

End of IBM Extension

Example 3:

This example illustrates the use of the character constant enclosed in apostrophes and the F, E, G, and I format codes.

Assume that the following statements are given:

```
76 FORMAT   ('0',F6.2,E12.3,G14.6,I5)
   WRITE    (6,76)A,B,C,N
```

and that the variables A, B, C, and N have the following values on successive executions of the WRITE statement:

A	B	C	N
034.40	123.380E+02	123.380E+02	031
031.1	1156.1E+02	123456789.	130
-354.32	834.621E-03	1234.56789	428
01.132	83.121E+06	123380.D+02	000

Then, the following lines are printed by successive executions of the WRITE statement:

```
Print
Column:      1       9             21              35

             v       v             v               v

              34.40   0.123E+05     12338.0          31

              31.10   0.116E+06   0.123457E 09      130

             ******   0.835E+00     1234.57         428

               1.13   0.831E+08   0.123380E 08        0
```

Explanation:

- The integer portion of the third value of A exceeds the format code specification, so asterisks are printed instead of a value. The fractional portion of the fourth value of A exceeds the format code specification, so the fractional portion is rounded.
- For the variable B, the decimal point is printed to the left of the first significant digit and only three significant digits are printed because of the format code E12.3. Excess digits are rounded off from the right.
- The values of the variable C are printed according to the format specification G14.6. The *d* specification, which in this case is 6, determines the number of digits to be printed and whether the number should be printed with a decimal exponent. Values greater than or equal to 0.1 and less than 1000000 are printed without a decimal exponent in this example. Thus, the first and third values have no exponent. The second and fourth values are greater than 1000000, so they are printed with an exponent.

L Format Code

The L format code **transmits logical variables**.

Input: The input field must consist of either zeros or blanks with an optional decimal point, followed by a T or 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 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 alphabetic or special character is given a unique internal code. Numeric characters are transmitted without alteration; they are not converted into a form suitable for computation. Thus, the A format code can be used for numeric fields, but not for numeric fields requiring arithmetic.

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 number 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 on the data set associated with data set reference number 6 from the variables A, B, and C in the format specified by FORMAT statement 26. The printed output for values of A, B, and C is as follows (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 two ways: following the H format code or enclosed in apostrophes. For example, the following FORMAT statements are equivalent.

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

25   FORMAT (' 1982 INVENTORY REPORT')
```

No item in the output list corresponds to the character constant. The constant is written directly from the FORMAT statement. (The FORMAT statement can contain other types of format code with corresponding variables in the I/O list.)

Input: Character constants cannot appear in a format used for input.

Output: The character constant from the FORMAT statement is written on the output file. (If the H format code is used, the *w* characters following the H are written. If apostrophes are used, the characters enclosed in apostrophes are written.) For example, the following statements:

```
8 FORMAT (14H0MEAN AVERAGE:, F8.4)

  WRITE (6,8) AVRGE
```

would write the following record if the value of AVRGE were 12.3456:

```
MEAN AVERAGE: 12.3456
```

The first character of the output data record in this example is the carrier control character for printed output. One line is skipped before printing, and the carrier control character does not appear in the printed line.

Note: If the character constant is enclosed in apostrophes, an apostrophe character in the data is represented by two successive apostrophes. For example, DON'T would be represented as 'DON''T'. The two successive apostrophes are counted as one character. A maximum of 255 characters can be specified in a character or a Hollerith constant.

X Format Code

The X format code **specifies a field of *w* characters to be skipped** on input or filled with blanks on output if the field was not previously filled. On output, an X format code does not affect the length of a record. For example, the following statements:

- Read the first ten characters of the input line into variable I.
- Skip over the next ten characters without transmission.
- 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 FORTRAN record at which the transfer of data is to begin.**

To illustrate the use of the T code, the following statements:

```
5   FORMAT (T40,'1981 STATISTICAL REPORT', T80,

  X  'DECEMBER',T1,'0PART NO. 10095')

    WRITE  (6,5)
```

print the following:

```
Print
Position:  1                  39                       79

           v                  v                        v
           PART NO. 10095     1981 STATISTICAL REPORT  DECEMBER
```

The T format code can be used in a FORMAT statement with any type of format code, as, for example, with FORMAT ('0',T40,I5).

Input: The T format code allows portions of a record to be processed more than once, possibly with different format codes.

Output: The record is assumed to be initially filled with blank characters, and the T format code can replace or skip characters. On output, a T format code does not affect the length of a record.

(For printed output, the first character of the output data record is a carrier control character and is not printed. Thus, for example, if T50,'Z' is specified in a FORMAT statement, a Z will be the 50th character of the output record, but it will appear in the 49th print position.)

TL and TR Format Codes: The TL and TR format codes specify how many characters left (TL) or right (TR) from the current character position the transfer of data is to begin. With TL format code, if the current position is less than or equal to the position specified with TL, the next character transmitted will be placed in position 1 (that is, the carrier control position).

The TL and TR format codes can be used in a FORMAT statement with any type of format code. On output, these format codes do not affect the length of a record.

Group Format Specification

The group format specification **repeats a set of format codes and controls the order in which the format codes are used.**

The group repeat count *a* is the same as the repeat indicator *a* that can be placed in front of other format codes. For example, the following statements are equivalent:

```
10    FORMAT  (I3,2(I4,I5),I6)

10    FORMAT  (I3,(I4,I5,I4,I5),I6)
```

Group repeat specifications control the order in which format codes are used, since 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. (See "Forms of a FORMAT Statement" on page 114.) 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. Since 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.

IBM Extension

The S, SP, and SS format codes also affect Q editing.

End of IBM Extension

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.

IBM Extension

The BN format code also affects Q editing during execution of an input statement.

End of IBM Extension

The BN format code has no effect during execution of an output statement.

Example:

The following program (containing both BN and BZ format code):

```
    READ   (9,100) R,S,I,J
    READ   (9,101) A,B,K,L
100 FORMAT (BZ,Q15.3,F7.2,I3,I7)
101 FORMAT (BN,Q15.3,F7.2,I3,I7)
    WRITE  (*,*) R,S,I,J,A,B,K,L
    STOP
    END
```

with the following input:

```
1.2         3.1  3       5
1.2         3.1  3       5
```

creates the following output:

```
1.19999980     3.10000038          300        5
1.19999980     3.10000038            3        5
```

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, and D editing during execution of an input statement.

IBM Extension

The BZ format code also affects Q editing during execution of an input statement.

End of IBM Extension

The BZ format code has no effect during execution of an output statement.

Slash Format Code

A slash **indicates the end of a VS FORTRAN 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 FORTRAN record formats. The first, third, etc., records are transmitted according to the format I3, F6.2 and the second, fourth, etc., 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 FORTRAN record formats. On output, it places a blank line between the line written with format 1X,10I5 and the line written with the format 1X,8E14.5.

For a file connected for direct access, when a slash is encountered, the record number is increased by one and the file is positioned at the beginning of the record that has that record number.

Colon Format Code

A colon **terminates format control** if there are no more items in the input/output list. The colon has no effect if there are more items in the input/output list.

Example:

Assume the following statements:

```
      ITABLE=10
      IELEM=0
         .
         .
         .
   10 WRITE(6,1000)ITABLE,IELEM
         .
         .
         .
      ITABLE=11
      IELEM=25
         .
         .
         .
      XMIN=-.37E1
      XMAX=.2495E3
         .
         .
         .
   20 WRITE(6,1000)ITABLE,IELEM,XMIN,XMAX
 1000 FORMAT('0 TABLE NUMBER',I5,:,'CONTAINS',I5,'ELEMENTS',:,
     1      /'MINIMUM VALUE:',E15.7,
     2      /'MAXIMUM VALUE:',E15.7)
```

The WRITE statement at statement number 10 generates the following:

```
TABLE NUMBER 10 CONTAINS 0 ELEMENTS
```

The WRITE statement at statement number 20 generates the following:

```
TABLE NUMBER 11 CONTAINS 25 ELEMENTS
MINIMUM VALUE: -.3700000E+01
MAXIMUM VALUE:  .2495000E+03
```

Reading Format Specifications at Object Time

VS FORTRAN provides for variable FORMAT statements by allowing a format specification to be read into a character array element or a character variable in storage. The data in the character array or variable may then be used as the format specification for subsequent input/output operations. The format specification may also be placed into the character array or variable by a DATA statement or an explicit specification statement in the source program. The following rules are applicable:

- The format specification must be a character array or character variable, even if the array size is only 1.
- 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 number are omitted. The parentheses surrounding the format codes are required.
- If a format code read at object time contains two consecutive apostrophes within a character field that is defined by apostrophes, it should be used for output only.

- Blank characters may precede the format specification, and character data may follow the right parenthesis that ends the format specification.

Example: Assume the following statements:

```
  DIMENSION C(5)
  CHARACTER*16 FMT
  READ(5,1)FMT
1 FORMAT (A)
  READ(5,FMT)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 line is read, converted, and stored in A,B, and the array C, according to the format codes 2E10.3, 5F10.8.

IBM Extension

Reading a FORMAT into a noncharacter array: Assume the following statements:

```
  DIMENSION FMT(16),C(5)
  READ(5,1) FMT
1 FORMAT(16A1)
  READ(5,FMT)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.

End of IBM Extension

List-Directed Formatting

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a constant, a null value, or one of the forms:

```
r*f
```

or

```
r*
```

where *r* is an unsigned, nonzero, integer constant. The *r**f* form is equivalent to *r* successive appearances of the constant *f*, and the *r** form is equivalent to *r* successive null values. Neither of these forms may contain embedded blanks, except where permitted within the constant *f*.

A *value separator* is one of the following:

- A comma, optionally preceded by one or more blanks and optionally followed by one or more blanks
- A slash, optionally preceded by one or more blanks and optionally followed by one or more blanks
- One or more blanks between two constants or following the last constant

Input: Input forms acceptable to format specifications for a given type are acceptable for list-directed formatting, except as noted below. The form of the input value must be acceptable for the type of the input list item. Blanks are never treated as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below. The end of a record has the effect of a blank, except when it appears within a character constant.

When the corresponding input list item is of real or double precision type, the input form is that of a numeric input field. A *numeric input field* is a field suitable for the F format code that is assumed to have no fractional digits, unless a decimal point appears within the field.

When the corresponding list item is of complex type, the input form consists of a left parenthesis, an ordered pair of numeric input fields separated by a comma, and a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of logical type, the input form must not include either slashes or commas among the optional characters permitted for the L format code.

When the corresponding list item is of character type, the input form consists of a nonempty string of characters enclosed in apostrophes. Each apostrophe within a character constant must be represented by two consecutive apostrophes without an intervening blank or the end of the record. Character constants may be continued from the end of one record to the beginning of the next record. The end of the record does not cause a blank or any other character to become part of the constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

For example, let *len* be the length of the list item, and let *w* be the length of the character constant. If *len* is less than or equal to *w*, the leftmost *len* characters of the constant are transmitted to the list item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the list item and the remaining *len-w* characters of the list item are filled with blanks. The effect is that the constant is assigned to the list item in a character assignment statement.

A null value is specified by having no characters between successive separators, by having no characters preceding the first value separator in the first record read by each execution of a list-directed input statement, or by the *r** form. A null value has no effect on the definition status by the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains

undefined. A null value may not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant. The end of a record following any other separator, with or without separating blanks, does not specify a null value.

A slash encountered as a value separator during execution of a list-directed input statement causes termination of execution of that input statement after the assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them.

All blanks in a list-directed input record are considered part of some value separator, except for the following:

- Blanks embedded in a character constant
- Embedded blanks surrounding the real or imaginary part of a complex constant
- Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma

Output: Except as noted, the form of the values produced is the same as that required for input. With the exception of character constants, the values are separated by one of the following:

- One or more blanks
- A comma, optionally preceded by one or more blanks and optionally followed by one or more blanks

VS FORTRAN may begin new records as necessary but, except for complex constants and character constants, the end of a record must not occur within a constant, and blanks must not appear within a constant.

Logical output constants are T for the value .TRUE. and F for the value .FALSE..

Integer output constants are produced with the effect of an I*w* edit descriptor for some reasonable value of *w*.

Real and double precision constants are produced with the effect of either an F format code or an E format code, depending on the magnitude *x* of the value and a range:

```
10**d1 ≤ x < 10**d2
```

where *d*1 and *d*2 are processor-dependent integer values. If the magnitude *x* is within this range, the constant is produced using 0PF*w.d*; otherwise, 1PE*w.d*E*e* is used. Reasonable processor-dependent values are used for each of the cases involved.

Complex constants are enclosed in parentheses, with a comma separating the real and imaginary parts. The end of a record may occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.

Character constants produced:

- Are not delimited by apostrophes
- Are not preceded or followed by a value separator
- Have each internal apostrophe represented externally by one apostrophe
- Have a blank character inserted at the beginning of any record that begins with the continuation of a character constant from the preceding record

If two or more successive values in an output record produced have identical values, the sequence of identical values is written.

Slashes, as value separators, and null values are not produced by list-directed formatting.

Each output record begins with a blank character to provide carrier control if the record is printed.

FUNCTION Statement

The FUNCTION statement identifies a function subprogram consisting of a FUNCTION statement followed by other statements that may include one or more RETURN statements. It is an independently written program that is executed wherever its name is referred to in another program.

Syntax

[*type*] FUNCTION *name* ([*arg*1 [, *arg*2] ...])

type
: is integer, real, double precision, complex, logical, or character[**len*1]

  where:

  ***len*1**
  : is the length specification. It is optional; if omitted, it is assumed to be 1. It may be an unsigned, nonzero, integer constant, an integer constant expression enclosed in parentheses, or an asterisk enclosed in parentheses. The expression can only contain integer constants; it must not include names of integer constants.

    If the name is of character type, all entry names must be of character type, and lengths must be the same. If one length is specified as an asterisk, all lengths must be specified as an asterisk.

name
: is the name of the function.

IBM Extension

name*len2
is the name of the function.

where:

****len2***
is a positive, nonzero, unsigned integer constant. It represents one of the permissible length specifications for its associated type. (See "Variable Types and Lengths" on page 25.) **len*2 is optional. It may be included only when *type* is specified. It must not be used when DOUBLE PRECISION or CHARACTER is specified.

End of IBM Extension

arg
is a dummy argument. It must be a variable or array name that may appear only once within the FUNCTION statement or dummy procedure name. If there is no argument, the parentheses must be present. (See "Dummy Arguments in a Function Subprogram" on page 140.)

A type declaration for a function name may be made by the predefined convention, by an IMPLICIT statement, by an explicit specification in the FUNCTION statement, or by an explicit type specification statement within the function subprogram. If the type of a function is specified in a FUNCTION statement, the function name must not appear in an explicit type specification statement.

The name of a function must not be in any other nonexecutable statement except a type statement.

Because the FUNCTION statement is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

The FUNCTION statement must be the first statement in the subprogram. The function subprogram may contain any FORTRAN statement except a SUBROUTINE statement, another FUNCTION statement, a BLOCK DATA statement, or a PROGRAM statement. If an IMPLICIT statement is used in a function subprogram, it must follow the FUNCTION statement and may only be preceded by another IMPLICIT statement, a PARAMETER, FORMAT, or ENTRY statement.

The name of the function (or one of the ENTRY names) must appear as a variable name in the function subprogram and must be assigned a value at least once during the execution of the subprogram in one of the following ways:

- As the variable name to the left of the equal sign in an arithmetic, logical, or character assignment statement
- As an argument of a CALL statement that will cause a value to be assigned in the subroutine referred to
- In the list of a READ statement within the subprogram

- As one of the parameters in an INQUIRE statement that is assigned a value within the subprogram
- As a DO- or implied DO-variable
- As the result of the IOSTAT specification in an I/O statement

The value of the function is the last value assigned to the name of the function when a RETURN or END statement is executed in the subprogram. For additional information on RETURN and END statements in a function subprogram, see "RETURN Statement" on page 222 and "END Statement" on page 94 .

The function subprogram may also use one or more of its arguments to return values to the calling program. An argument so used must appear:

- On the left side of an arithmetic, logical, or character assignment statement
- In the list of a READ statement within the subprogram
- As an argument in a function reference that is assigned a value by the function referred to
- As an argument in a CALL statement that is assigned a value in the subroutine referred to
- As one of the parameters in an INQUIRE statement

The dummy arguments of the function subprogram (for example, *arg*1, *arg*2, *arg*3,..., *arg*n) 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 30.

If the predefined convention is not correct, the function name must be typed in the program units that refer to it. The type and length specifications of the function name in the function reference must be the same as those of the function name in the FUNCTION statement.

Except in a character assignment statement, the name of a character function whose length specification is an asterisk must not be the operand of a concatenation operation.

The length specified for a character function in the program unit that refers to the function must agree with the length specified in the subprogram that specifies the function. There is always agreement of length if the asterisk is used in the referenced subprogram to specify the length of the function.

Actual Arguments in a Function Subprogram

The actual arguments in a function reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced function. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type.

If an actual argument is of type character, the associated dummy argument must be of type character and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument is greater than the length of an associated dummy argument, the leftmost characters of the actual argument are associated with the dummy argument.

An actual argument in a function reference must be one of the following:

- An array name
- An intrinsic function name
- An external procedure name
- A dummy argument name
- An expression, except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses (unless the operand is the name of a constant).

For an entry point in a function subprogram, see "ENTRY Statement" on page 97.

Dummy Arguments in a Function Subprogram

The dummy arguments of a function subprogram appear after the function name and are enclosed in parentheses. They are replaced at the time of invocation by the actual arguments supplied in the function reference.

Dummy arguments must adhere to the following rules:

- None of the dummy argument names may appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, SAVE, INTRINSIC, or NAMELIST statement, except as NAMELIST or common block names, in which case the names are not associated with the dummy argument names.
- A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, ENTRY or statement function definition in the same program unit.
- The dummy arguments must correspond in number, order, and type to the actual arguments.

- If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, an array element, a substring, or an array. A constant, name of constant, subprogram name, or expression should not be written as an actual argument unless the programmer is certain that the corresponding dummy argument has not been assigned a value in the subprogram.

- A referenced subprogram cannot assign new values to dummy arguments that are associated with other dummy arguments within the subprogram or with variables in the common block.

Valid Examples

1. Definition of function subprogram SUFFIX:

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

   Use of function subprogram SUFFIX:

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

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

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

   Use of function subprogram CUBE. Functions defined without any dummy arguments must be invoked with the null parentheses.

```
REAL*16 A,X
COMMON /COM1/ A
A = 1.6

X = CUBE()
```

3. Function IADD illustrates assigning a value to the function name (in this case, IADD) by means of an argument of a CALL statement.

```
FUNCTION IADD( M )
...
CALL SUBA (IADD, M)
RETURN
END
```

   Definition of subroutine SUBA:

```
SUBROUTINE SUBA (J,K)
J = 10 + K
RETURN
END
```

4. Function IREAD illustrates assigning a value to the name of a function (in this case, IREAD) by means of an I/O list of a READ statement within the function definition.

```
FUNCTION IREAD ()
READ *, IREAD
RETURN
END
```

5. Function SUM illustrates the use of adjustable dimensions.

```
   INTEGER FUNCTION SUM(ARRY, M, N)
   INTEGER M, N, ARRY(M, N)

   SUM = 0
   DO 10 I = 1, M
   DO 10 J = 1, N
10 SUM = SUM + ARRY(I,J)
   RETURN
   END
```

Use of function subprogram SUM:

```
DIMENSION IARRAY(20,30)
INTEGER SUM
...
IVAR = SUM(IARRAY, 20, 30)
```

Invalid Examples

Assume the following function definition:

```
REAL FUNCTION BAD(ARG)

IF ( ARG .EQ. 0.0 ) ARG = 1.0
BAD = 123.4/ARG

RETURN
END
```

The following use of BAD is illegal, because the actual argument is an expression, and BAD may assign a value to its dummy argument.

```
X = BAD( 6.0 * X )
```

The following use of BAD is also illegal, because the actual argument is a constant.

```
X = BAD( 12.3 )
```

GO TO Statements

GO TO statements transfer control to an executable statement in the program unit. There are three GO TO statements:

- Assigned GO TO statement
- Computed GO TO statement
- Unconditional GO TO statement

Assigned GO TO Statement

The assigned GO TO statement transfers control to the statement numbered *stn*1, *stn*2, *stn*3 ..., depending on whether the current assignment of *i* is *stn*1, *stn*2, *stn*3 ..., respectively. (See "ASSIGN Statement" on page 59.)

Syntax

GO TO *i* [[,] (*stn*1 [,*stn*2] [,*stn*3] ...)]

i

is an integer variable (not an array element) of length 4 that has been assigned a statement number by an ASSIGN statement.

stn

is the number of an executable statement in the same program unit as the assigned GO TO statement.

The list of statement numbers, that is, (*stn*1, *stn*2, *stn*3 ...), is optional. If omitted, the preceding comma must be omitted. If the list of statement numbers is specified, the preceding comma is optional. The statement number assigned to *i* must be one of the statement numbers in the list. The statement number may appear more than once in the list.

The ASSIGN statement that assigns the statement number to *i* must appear in the same program unit as the assigned GO TO statement that is using this statement number.

For example, in the statement:

```
GO TO N, (10, 25, 8)
```

If the current assignment of the integer variable N is statement number 8, then the statement numbered 8 is executed next. If the current assignment of N is statement number 10, the statement numbered 10 is executed next. If N is assigned statement number 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 number 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, then it must be assigned a statement number in the subprogram, and also used in an assigned GO TO in that subprogram. An integer variable used as an actual argument in a subprogram reference may not be used in an assigned GO TO in the invoked subprogram unless it is redefined in the subprogram.

Any executable statement immediately following the assigned GO TO statement should have a statement number; 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 numbered *stn*1, *stn*2, or *stn*3,... depending on whether the current value of *m* is 1, 2, or 3,... respectively.

> **Syntax**
>
> GO TO (*stn*1 [, *stn*2] [, *stn*3] ...) [,] *m*

stn
: is the number of an executable statement in the same program unit as the computed GO TO statement. The same number may appear more than once within the parentheses.

m
: is an integer expression. The comma before *m* is optional. If the value of *m* is outside the range $1 \leq m \leq n$, where n is the number of statement numbers, the next statement is executed.

A computed GO TO statement may terminate the range of a DO.

Example:

```
171  GO TO(172,173,174,173) INT(A)
172  A = A + 1.0
     GO TO 174
173  A = A + 1.0
174  CONTINUE
```

Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the statement specified by the statement number. Every subsequent execution of this GO TO statement results in a transfer to that same statement.

> **Syntax**
>
> GO TO *stn*

stn
: is the number 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 number; 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

  END IF
  ELSE
  ELSE IF
- Logical IF

Arithmetic IF Statement

The arithmetic IF statement transfers control to the statement numbered *stn*1, *stn*2, or *stn*3 when the value of the arithmetic expression (*m*) is less than, equal to, or greater than zero, respectively. The same statement number may appear more than once within the same IF statement.

Syntax

IF (*m*) *stn*1, *stn*2, *stn*3

m

is an arithmetic expression of any type except complex.

stn

is the number 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 number; 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 (*stn*) 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 (*stn*).

n2

is the number of END IF statements in the program unit up to, but not including, the statement (*stn*).

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 ELSE IF statement cannot terminate the range of a DO. Execution of an END IF statement has no effect.

Valid Examples:

The following is the general form of a single alternative block IF statement (in other words, no ELSE or ELSE IF statements are in the IF-block).

```
     IF ( m ) THEN
C
C      EXECUTION SEQUENCE WHEN THE VALUE OF m IS TRUE
C
     ...
     ENDIF
C
C      IF m IS FALSE, EXECUTION CONTINUES HERE
C
     ...
```

The following is an example of a single alternative IF.

```
     IF ( INDEX .EQ. 0) THEN
        PRINT *, 'KEY NOT FOUND'
        INDEX = - 1
     ENDIF
     ...
```

ELSE Statement

The ELSE statement is executed if the preceding block IF or ELSE IF condition is evaluated as FALSE. Normal execution sequence continues.

Syntax

ELSE

An ELSE-block consists of all the executable statements after the ELSE statement up to, but not including, the next END IF statement that has the same IF-level as the ELSE statement. An ELSE-block may be empty.

Within an IF-block, you can have only one ELSE.

Transfer of control into an ELSE-block from outside the ELSE-block is prohibited. The statement number, 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 Examples

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 number (*stn*), 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 ELSE IF statement cannot terminate the range of a DO.

Valid Examples:

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 if both *m* and *m1* are false.

```
...
```

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 = 'GREEN'
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 or a block IF, ELSE IF, ELSE, or END IF statement.

IBM Extension

st may not be a TRACE ON, TRACE OFF, INCLUDE, or DISPLAY statement.

End of IBM Extension

The statement *st* must not have a statement number.

The execution of a function reference in *m* is permitted to affect entities in the statement *st*.

The logical IF statement containing *st* may have a statement number. If a logical IF statement terminates the end of a DO loop, it may not contain a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

Example:

```
   IF(A.LE.0.0) GO TO 25
   C = D + E
   IF (A.EQ.B) ANSWER = 2.0*A/C
   F = G/H
25 W = X**Z
      .
      .
      .
```

IMPLICIT Type Statement

The IMPLICIT type statement specifies the type and length of all variables, arrays, and user-supplied functions whose names begin with a particular letter. It may be used to change or confirm implicit typing.

Syntax

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

type

is character[**len*1], complex, double precision, integer, logical, or real

where:

***len*1**

can be an unsigned, nonzero, integer constant or a positive integer constant expression enclosed in parentheses. It is optional.

If *len*1 is not specified, the length is one.

IBM Extension

type

is complex[**len*2], integer[**len*2], logical[**len*2], or real[**len*2]

where:

***len*2**

can be a positive, nonzero, unsigned, integer constant. It represents one of the permissible length specifications for its associated type. It is optional.

End of IBM Extension

a

is a single alphabetic character or a range of characters drawn from the set A, B,..., Z. The range is denoted by the first and last characters of the range separated by a minus sign (for example, A-D).

IBM Extension

The alphabetic character *a* can also be the currency symbol ($). The currency symbol ($) follows the letter Z. Thus, the range Y-$ is the same as Y,Z,$.

End of IBM Extension

The IMPLICIT specification statement can only be preceded by a PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA, PARAMETER, ENTRY, or FORMAT statement, or another IMPLICIT statement. The IMPLICIT specification statement declares the type of the variables and user-supplied functions appearing in this program (that is, integer, real, complex, logical, or character) by specifying that names beginning with certain designated letters are of a certain type. Furthermore, the IMPLICIT statement allows the programmer to declare the number of bytes to be allocated for each in the group of specified variables.

The CHARLEN compiler option may be specified to set 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.

The type and length associated with a letter or a range of letters must not conflict with the type or length given previously to the same letters in the same IMPLICIT statement, in a different IMPLICIT statement or in a PARAMETER statement. Type specification by an IMPLICIT statement may be overridden or confirmed for any particular variable, array, name of a constant, external function, or statement function name by the appearance of that name in an explicit type specification statement.

(See "Type Declaration by the Predefined Specification" on page 27.)

Note: An IMPLICIT statement has no effect on names of VS FORTRAN-supplied (intrinsic) functions.

Valid IMPLICIT Statements:

```
IMPLICIT INTEGER(A-H), REAL(I-K), LOGICAL(L,M,N)

IMPLICIT COMPLEX(C-F)
```

IBM Extension

```
IMPLICIT INTEGER(W-$)
```

All names beginning with W, X, Y, Z, and $ are considered integers of length 4 bytes.

End of IBM Extension

IBM Extension

INCLUDE Statement

The INCLUDE statement is a compiler directive. It inserts a specified statement or a group of statements into a program unit.

A function called conditional INCLUDE provides a means for selectively activating INCLUDE statements within the VS FORTRAN source during compilation. The included files are specified by means of the CI compiler option. For more information about the CI compiler option and how to use the INCLUDE statement, see *VS FORTRAN Programming Guide.*

Syntax

INCLUDE (*name*) [*n*]

name
is the name of a group of one or more VS FORTRAN source statements to be inserted into the source program being compiled. The group must reside in a library known to the VS FORTRAN compiler.

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 CI list. The range of *n* is 1 to 255.

The following rules apply to the INCLUDE statement:

- INCLUDE is a compiler directive statement only.
- The INCLUDE statement may not be continued.
- No replacement or editing is done.
- The inserted group may contain any VS FORTRAN source statements, including other INCLUDE statements.
- An INCLUDE of a group may not contain an INCLUDE statement that refers to a currently open INCLUDE group (that is, recursion is not permitted).

- Multiple INCLUDE statements may appear in the original source program.
- INCLUDE statements may appear anywhere in a source program before the END statement, except as the trailer of a logical IF statement. An END statement may be part of the included group.
- The VS FORTRAN statements in the group being included must be in the same form as the source program being compiled; that is, fixed form or free form.
- After the inclusion of all groups, the resulting VS FORTRAN program must follow all VS FORTRAN rules for sequencing of statements.

End of IBM Extension

INQUIRE Statement

An INQUIRE statement supplies information about properties of a particular named external file or of the connection to a particular external unit. This information is determined by the VS FORTRAN I/O statements that have been processed, not by testing for operating system information. In other words, specification of INQUIRE is limited to currently or previously opened files.

There are two forms of the INQUIRE statement:

- Inquire by **file name**
- Inquire by **unit number**

A file can be queried about its existence, its unit number, its name, the kind of processing it can be opened for, whether it has in fact been opened, whether it is formatted or unformatted, and how blanks are to be interpreted.

In addition, a file opened for direct access can be queried about its record length or its next record number. A file opened for keyed access can be queried about:

- The way it was opened (for reading, writing, or both)
- Which of multiple keys is in use, and its length and position
- The value of the last key used in a READ, WRITE, REWRITE, or BACKSPACE operation
- The length of the last record processed by a READ, WRITE, REWRITE, or BACKSPACE operation

The INQUIRE statement can be executed before, while, or after a file is connected to a unit. All values assigned by the INQUIRE statement are those that are current at the time the statement is executed. All value assignments are done according to the rules for assignment statements. No error is given if the value is truncated because the receiving field is too small to contain it all.

INQUIRE by File Name

This INQUIRE statement supplies information about a file. When this statement is executed, the file specified by *fn* may or may not be connected to a unit. If the file is connected to a unit, the file may or may not exist. (For example, an output unit may be connected to a file, but no output has been written.)

Syntax

INQUIRE (FILE=*fn* [, ERR=*stn*] [, IOSTAT=*ios*] [, EXIST=*exs*]

[, OPENED=*opn*] [, NAMED=*nmd*]

[, NAME=*nam*] [, SEQUENTIAL=*seq*]

[, DIRECT=*dir*] [, KEYED=*kyd*]

[, FORMATTED=*fmt*] [, UNFORMATTED=*unf*]

[, NUMBER=*num*] [, ACCESS=*acc*] [, FORM=*frm*]

[, RECL=*rcl*] [, NEXTREC=*nxr*] [, BLANK=*blk*]

[, ACTION=*acc*] [, WRITE=*wri*]

[, READ=*ron*] [, READWRITE=*rwr*]

[, KEYID=*kid*] [, KEYLENGTH=*kle*]

[, KEYSTART=*kst*] [, KEYEND=*ken*]

[, LASTKEY=*lky*] [, LASTRECL=*lrl*])

All parameters except FILE=*fn* are optional.

FILE=*fn*
is required. *fn* is the reference to a file and *must* be preceded by FILE=. It is a character expression. Its value, when any trailing blanks are removed, must be 1 to 7 characters, the first one being one of the 26 alphabetic characters, and the other six being of the 26 alphabetic or the 10 numeric characters. It must be the name of the file being inquired about and must be known to the program.

ERR=*stn*
stn is the number of an executable statement in the same program unit as the INQUIRE statement. If an error occurs, control is transferred to *stn*.

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

EXIST=*exs*
exs is a logical variable or logical array element of length 4. It is assigned the value true if the file by the specified name exists; otherwise, it is assigned the value false. No value is assigned if an error has occurred.

OPENED=*opn*
opn is a logical variable or a logical array element of length 4. It is assigned the value true if the file specified is connected to a unit; otherwise, it is assigned the value false. No value is assigned if an error has occurred.

The File Exists: The following parameters have a value only if the file being inquired about exists; that is, *exs* has the value true. These parameters are all optional.

NAMED=*nmd*
nmd is a logical variable or a logical array element of length 4. If the file has a name (*fn*), *nmd* is assigned the value true; otherwise, it is assigned the value false.

NAME=*nam*
nam is a character variable or character array element. If the file has a name (*fn*), *nam* is assigned the value of *name*. *name* is not necessarily the same as the name in the FILE parameter (*fn*).

SEQUENTIAL=*seq*
seq is a character variable or a character array element. It is assigned the value YES if the file can be connected for sequential access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for sequential access.

DIRECT=*dir*
dir is a character variable or a character array element. It is assigned the value YES if the file can be connected for direct access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for direct access.

IBM Extension

KEYED=*kyd*
kyd is a character variable or a character array element. It is assigned the value YES if the file can be connected for keyed access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for keyed access.

End of IBM Extension

FORMATTED=*fmt*
fmt is a character variable or character array element. It is assigned the value YES if the file can be connected for formatted input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for formatted input/output.

UNFORMATTED=*unf*

unf is a character variable or character array element. It is assigned the value YES if the file can be connected for unformatted input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for unformatted input/output.

The file is Connected to an Existing Unit: The following parameters have a value only if the file exists (*exs* has the value true), and if the file is connected to a unit (*opn* has the value true). These parameters are all optional.

NUMBER=*num*

num is an integer variable or integer array element of length 4. It is assigned the value of the external unit connected to the file.

ACCESS=*acc*

acc is a character variable or character array element. If there is a name *fn*, *acc* is assigned a value (SEQUENTIAL, DIRECT, or KEYED) associated with the connection of the external file.

FORM=*frm*

frm is a character variable or character array element. It is assigned the value FORMATTED if the file is connected for formatted input/output; UNFORMATTED if the file is connected for unformatted input/output.

The File is Connected for Direct Access I/O: The following parameters have a value only if the file exists (*exs* has the value true), and if the file is connected for direct access (*acc*=DIRECT). These files are all optional. The file must have been explicitly opened.

RECL=*rcl*

rcl is an integer expression of length 4. Its value is 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.

NEXTREC=*nxr*

nxr is an integer variable or integer array element of length 4. It is assigned the value n+1, where n is the record number of the last record read or written on the direct access file. If the file is connected, but no records have been read or written since the connection, *nxr* is assigned the value 1.

The File is Connected for Formatted I/O: The following parameter has a value only if the file exists (*exs* has the value true) and if the file is connected for formatted input/output (*frm* has the value FORMATTED). The parameter is optional.

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.

The File is Connected for Keyed Access I/O: The following parameters have a value only if the file exists (*exs* has the value true) and if the file is connected for keyed access (*acc* has the value KEYED). These parameters are all optional. The file must have been explicitly opened.

IBM Extension

ACTION=*act*
act is a character variable or character array element that is assigned one of the following values:

WRITE	If the file was opened to load records into an empty keyed file
READ	If the file was opened only to retrieve records
READWRITE	If the file was opened to allow retrieval and update operations

WRITE=*wri*
wri is a character variable or character array element that is assigned the value YES if the keyed file was opened to load records into the file; otherwise, it is assigned the value NO.

READ=*ron*
ron is a character variable or character array element that is assigned the value YES if the keyed file was opened only for retrieval; otherwise, it is assigned the value NO.

READWRITE=*rwr*
rwr is a character variable or character array element that is assigned the value YES if the keyed file was opened to allow retrieval and update operations; otherwise, it is assigned the value NO.

KEYID=*kid*
kid is an integer variable or integer array element of length 4. It is assigned the relative position of a *start-end* pair in the list of such pairs in the KEYS parameter of the OPEN statement. If the OPEN statement has no KEYS parameter, a value of 1 is assigned.

KEYLENGTH=*kle*
kle is an integer variable or integer array element of length 4. It is assigned the length of the key currently in use.

KEYSTART=*kst*
kst is an integer variable or integer array element of length 4. It is assigned the position of the leftmost character in the record of the key currently in use.

KEYEND=*ken*
ken is an integer variable or integer array element of length 4. It is assigned the position of the rightmost character in the record of the key currently in use.

LASTKEY=*lky*

lky is a variable array element of any data type. It is assigned the value of the key of the last keyed file record that was retrieved with a READ statement, written with a WRITE or REWRITE statement, deleted with a DELETE statement, or positioned to the beginning with a BACKSPACE statement. To receive the full key, *lky* must be at least as long as the key. If it is shorter, the value of the key is truncated on the right to make it the same length. If *lky* is longer, binary zeros are added to the right of the value to make it the same length. The assigned value is not meaningful if the last input/output operation was unsuccessful or was a REWIND, OPEN, or CLOSE operation.

LASTRECL=*lrl*

lrl is an integer variable or integer array element of length 4. It is assigned the length of the last keyed file record that was retrieved with a READ statement, written with a WRITE or REWRITE statement, deleted with a DELETE statement, or positioned to the beginning with a BACKSPACE statement. The assigned value is not meaningful if the last input/output operation was unsuccessful or was a REWIND, OPEN, or CLOSE operation.

End of IBM Extension

The parameters can be entered in any order. Each parameter cannot appear more than once in an INQUIRE statement. The same variable or array element cannot be specified for more than one parameter in the same INQUIRE statement.

Valid INQUIRE Statements:

```
INQUIRE (FILE=DDNAME, IOSTAT=IOS, EXIST=LEX, OPENED=LOD,
         NAMED=LNMD, NAME=FN, SEQUENTIAL=SEQ, DIRECT=DIR,
         FORMATTED=FMT, UNFORMATTED=UNF, ACCESS=ACC, FORM=FRM,
         NUMBER=INUM, RECL=IRCL, NEXTREC=INR, BLANK=BLNK)

INQUIRE (FILE='FT16K01',LASTRECL=RECL)
```

INQUIRE by Unit Number

This INQUIRE statement supplies information about an input/output unit.

A unit can be queried about its existence and its connection to a file. If it is connected to a file, the inquiry is being made about the connection and the file connected. When this statement is executed, the unit specified by *un* may or may not be connected to a file. If the unit is connected to a file, the file may or may not exist. For example, an output unit may be connected to a file but no output has been written.

INQUIRE

Syntax

INQUIRE ([UNIT=]*un*[, ERR=*stn*] [, IOSTAT=*ios*] [, EXIST=*exs*]

[, OPENED=*opn*] [, NAMED=*nmd*]

[, NAME=*nam*] [, SEQUENTIAL=*seq*]

[, DIRECT=*dir*] [, KEYED=*kyd*]

[, FORMATTED=*fmt*] [, UNFORMATTED=*unf*]

[, NUMBER=*num*] [, ACCESS=*acc*] [, FORM=*frm*]

[, RECL=*rcl*] [, NEXTREC=*nxr*] [, BLANK=*blk*]

[, ACTION=*acc*] [, WRITE=*wri*]

[, READ=*ron*] [, READWRITE=*rwr*]

[, KEYID=*kid*] [, KEYLENGTH=*kle*]

[, KEYSTART=*kst*] [, KEYEND=*ken*]

[, LASTKEY=*lky*] [, LASTRECL=*lrl*])

All parameters except UNIT=*un* are optional.

UNIT=*un*

un is the reference number of an I/O unit. It is an integer expression of length 4 whose value (zero or positive) represents the unit number that is being queried.

It is required and can optionally be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the INQUIRE statement. If an error occurs during the writing of an error message, control is transferred to *stn*.

IOSTAT=*ios*

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

EXIST=*exs*

exs is a logical variable or logical array element of length 4. It is assigned to value true if the specified unit exists and is known to the program unit. If neither of these conditions is met, *exs* is assigned the value false.

OPENED=*opn*

opn is a logical variable or logical array element of length 4. It is assigned the value true if the file specified is connected to a unit; otherwise, it is assigned the value false.

The Unit is Connected to an External File: The following parameters have a value only if the unit exists (*exs* has the value true) and the unit is connected to an external file (*opn* has the value true). These parameters are all optional.

NAMED=*nmd*

nmd is a logical variable or a logical array element of length 4. It is assigned the value true if the file connected to the unit has a name; otherwise, it is assigned the value false.

NAME=*nam*

nam is a character variable or character array element. If the file connected to the unit has a name, it is assigned the value of the name of that file. If the file is unnamed, a default name is assigned.

SEQUENTIAL=*seq*

seq is a character variable or a character array element. It is assigned the value YES if the file can be connected for sequential access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for sequential access.

DIRECT=*dir*

dir is a character variable or a character array element. It is assigned the value YES if the file can be connected for direct access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for direct access.

IBM Extension

KEYED=*kyd*

kyd is a character variable or a character array element. It is assigned the value YES if the file can be connected for keyed access input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for keyed access.

End of IBM Extension

FORMATTED=*fmt*

fmt is a character variable or character array element. It is assigned the value YES if the file can be connected for formatted input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for formatted input/output.

UNFORMATTED=*unf*

unf is a character variable or character array element. It is assigned the value YES if the file can be connected for formatted input/output; NO if it cannot; and UNKNOWN if it is not possible to determine whether the file can be connected for formatted input/output.

NUMBER=*num*

num is an integer variable or integer array element of length 4. Its value is the value of *un*.

ACCESS=*acc*

acc is a character variable or character array element. If there is a name *fn*, *acc* is assigned a value (SEQUENTIAL, DIRECT, or KEYED) associated with the connection of the external file.

FORM=*frm*

frm is a character variable or character array element. *frm* is assigned the value FORMATTED if the file is connected for formatted input/output; UNFORMATTED if the file is connected for unformatted output.

The Unit is Connected to an External File for Direct Access I/O: The following parameters have a value only if the unit exists (*exs* has the value true) and is connected to an external file for direct access input/output (*acc* has the value DIRECT). These parameters are all optional.

RECL=*rcl*

rcl is an integer variable or integer array element of length 4. It is assigned the value of the record length of the direct access file. The length is measured in characters for files consisting of formatted records and in bytes for files consisting of unformatted records.

NEXTREC=*nxr*

nxr is an integer variable or integer array element of length 4. It is assigned the value n+1 where n is the record number of the last record read or written on the direct access file. If the file is connected, but no records have been read or written since the connection, *nxr* is assigned the value 1.

BLANK=*blk*

blk is a character variable or character array element. It is assigned the value NULL if blanks in arithmetic input fields are treated as blanks; ZERO if they are treated as zeros.

The Unit Is Connected to an External File for Keyed Access I/O: The following parameters have a value only if the unit exists (*exs* has the value true) and is connected to an external file for keyed access (*acc*=KEYED). These parameters are all optional. The file must have been explicitly opened.

IBM Extension

ACTION=*act*

act is a character variable or character array element that is assigned one of the following values:

WRITE	If the file was opened to load records into an empty keyed file.
READ	If the file was opened only to retrieve records.
READWRITE	If the file was opened to allow retrieval and update operations.

WRITE=*wri*
: *wri* is a character variable or character array element that is assigned the value YES if the keyed file was opened to load records into the file; otherwise, it is assigned the value NO.

READ=*ron*
: *ron* is a character variable or character array element that is assigned the value YES if the keyed file was opened only for retrieval; otherwise, it is assigned the value NO.

READWRITE=*rwr*
: *rwr* is a character variable or character array element that is assigned the value YES if the keyed file was opened to allow retrieval and update operations; otherwise, it is assigned the value NO.

KEYID=*kid*
: *kid* is an integer variable or integer array element of length 4. It is assigned the relative position of a *start-end* pair in the list of such pairs in the KEYS parameter of the OPEN statement. If the OPEN statement has no KEYS parameter, a value of 1 is assigned.

KEYLENGTH=*kle*
: *kle* is an integer variable or integer array element of length 4. It is assigned the length of the key currently in use.

KEYSTART=*kst*
: *kst* is an integer variable or integer array element of length 4. It is assigned the position of the leftmost character in the record of the key currently in use.

KEYEND=*ken*
: *ken* is an integer variable or integer array element of length 4. It is assigned the position of the rightmost character in the record of the key currently in use.

LASTKEY=*lky*
: *lky* is a variable array element of any data type. It is assigned the value of the key of the last keyed file record that was retrieved with a READ statement, written with a WRITE or REWRITE statement, deleted with a DELETE statement, or positioned to the beginning with a BACKSPACE statement. To receive the full key, *lky* must be at least as long as the key. If it is shorter, the value of the key is truncated on the right and to make it the same length. If *lky* is longer, binary zeros are added to the right of the value to make it the same length. The assigned value is not meaningful if the last input/output operation was unsuccessful or was a REWIND, OPEN, or CLOSE operation.

LASTRECL=*lrl*
: *lrl* is an integer variable or integer array element of length 4. It is assigned the length of the last keyed file record that was retrieved with a READ statement, written with a WRITE or REWRITE statement, deleted with a DELETE statement, or positioned to the beginning with a BACKSPACE statement. The assigned value is not meaningful if the last input/output

operation was unsuccessful or was a REWIND, OPEN, or CLOSE operation.

End of IBM Extension

The parameters can be entered in any order unless UNIT=*un* is omitted. If omitted, *un*, as described under UNIT=*un*, must be first.

Valid INQUIRE Statements:

```
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)
```

INTRINSIC Statement

The INTRINSIC statement identifies a name as representing a FORTRAN-supplied procedure (function or subprogram) and permits a specific intrinsic function name to be used as an actual argument.

Syntax

INTRINSIC *name*1 [, *name*2] ...

name
is the generic or specific name of a VS FORTRAN intrinsic function.

The INTRINSIC statement is a specification statement and must precede statement function definitions and all executable statements.

Intrinsic functions are those functions known to the compiler. Intrinsic function names are either generic or specific. A generic name does not have a type, unless it is also a specific name.

Generic names simplify referring to intrinsic functions because the same function name may be used with more than one type of argument. Only a specific intrinsic function name may be used as an actual argument when the argument is an intrinsic function.

See Chapter 8, "Mathematical, Character, and Bit Subprograms" on page 307, for the complete list of intrinsic function names and usage information for each function.

Appearance of a name in an INTRINSIC statement declares that name to be an intrinsic function name. If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit.

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

```
AMAX0        INT
AMAX1        LGE
AMIN0        LGT
AMIN1        LLE
CHAR         LLT
DMAX1        MAX0
DMIN1        MAX1
FLOAT        MIN0
ICHAR        MIN1
IDINT        REAL
IFIX         SNGL
```

IBM Extension

```
CMPLX        QCMPLX
DBLE         QEXT
DBLEQ        QEXTD
DCMPLX       QFLOAT
DFLOAT       QMAX1
DREAL        QMIN1
HFIX         QREAL
IQINT        SNGLQ
```

End of IBM Extension

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 a VS FORTRAN intrinsic function appears in an explicit specification statement, the type must confirm its associated type.

If the name of a VS FORTRAN intrinsic function appears in the dummy argument list of a subprogram, that name is not considered as the name of a VS FORTRAN intrinsic function in that program unit.

Logical IF Statement

See "IF Statements" on page 145.

Logical Type Statement

See "Explicit Type Statement" on page 103.

IBM Extension

NAMELIST Statement

The NAMELIST statement specifies one or more lists of names for use in READ and WRITE statements.

Syntax

NAMELIST /*name*1/ *list*1 /*name*2/ *list*2 ...

name

is a NAMELIST name. It is a name, enclosed in slashes, that must not be the same as a variable or array name.

list

is of the form *a*1, *a*2,..., *an*

where:

a

is a variable name or an array name.

The list of variables or array names belonging to a NAMELIST name ends with a new NAMELIST name enclosed in slashes or with the end of the NAMELIST statement. A variable name or an array name may belong to one or more NAMELIST lists.

Neither a dummy variable nor a dummy array name may appear in a NAMELIST list.

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

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

The rules for input/output conversion of NAMELIST data are the same as the rules for data conversion described in "General Rules for Data Conversion" on page 111. The NAMELIST data must be in a special form, described in "NAMELIST Input Data" on page 167.

NAMELIST Input Data

Input data must be in a special form in order to be read using a NAMELIST list. The first character in each record to be read must be blank. The second character in the first record of a group of data records must be an ampersand (&) immediately followed by the NAMELIST name. The NAMELIST name must be followed by a blank and must not contain any embedded blanks. This name is followed by data items separated by commas. (A comma after the last item is optional.) The end of a data group is signaled by &END.

The form of the data items in an input record is:

- Name = Constant
  - The name may be an array element name or a variable name.
  - The constant may be integer, real, complex, logical, or character. (If the constants are logical, they may be in the form T or .TRUE. and F or .FALSE.; if the constants are characters, they must be included between apostrophes.)
  - Subscripts must be integer constants.
- Array Name = Set of Constants (separated by commas)
  - The *set of constants* consists of constants of the type integer, real, complex, logical, or character.
  - The number of constants must be less than or equal to the number of elements in the array.
  - Successive occurrences of the same constant can be represented in the form *c***constant*, where *c* is a nonzero integer constant specifying the number of times the constant is to occur.

The variable names and array names specified in the input file must appear in the NAMELIST list, but the order is not significant. A name that has been made equivalent to a name in the input data cannot be substituted for that name in the NAMELIST list. The list can contain names of items in COMMON but must not contain dummy argument names.

Each data record must begin with a blank followed by a complete variable or array name or constant. Embedded blanks are not permitted in names or constants. Trailing blanks after integers and exponents are treated as zeros.

Examples:

All records have a blank in column 1, and begin in column 2.

```
&NAM1 I(2,3)=5,J=4,B=3.2
    .
    .
    .
A(3)=4.0,L=2,3,7*4,&END
```

where NAM1 is defined in a NAMELIST statement as:

```
NAMELIST /NAM1/A,B,I,J,L
```

and assuming that A is a 3-element array and I and L are 3X3 element arrays.

NAMELIST Output Data

When output data is written using a NAMELIST list, it is written in a form that can be read using a NAMELIST list. 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.

End of IBM Extension

OPEN Statement

An OPEN statement may be used to:

- Connect an existing file to a unit.
- Create a file that is preconnected.
- Create a file and connect it to a unit.
- Change certain identifiers of a connection between a file and a unit.

For more information on how to use the OPEN statement with your operating system, see *VS FORTRAN Programming Guide*.

Syntax

OPEN ([UNIT=]*un* [, ERR=*stn*] [, STATUS=*sta*] [, FILE=*fn*]

[, ACCESS=*acc*] [, BLANK=*blk*] [, FORM=*frm*]

[, IOSTAT=*ios*] [, RECL=*rcl*]

[, ACTION=*act*] [, PASSWORD=*pwd*]

[, KEYS=(*start*:*end* [, *start*:*end*] ...)])

Each of the parameters of the OPEN statement can appear only once. The unit specifier (*un*) must appear. All value assignments are made according to the rules for assignment statements.

Before the OPEN statement is executed, the I/O unit specified by *un* may be either connected or not connected to an external file.

OPEN is required for direct-access and VSAM files. It is optional for sequential files and invalid for internal files.

UNIT=*un*

un is the reference number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the OPEN statement. If an error is detected, control is transferred to *stn*.

STATUS=*sta*

is optional. *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.

If the status of the external file is specified as:

- NEW, FILE=*fn* may be specified.
- OLD, FILE=*fn* may be specified.
- SCRATCH, FILE=*fn* must not be specified.
- UNKNOWN, FILE=*fn* is optional.

FILE=*fn*

is optional. *fn* is the reference to a file and *must* be preceded by FILE=. It is a character expression. Its value, when any trailing blanks are removed, must be 1 to 7 characters, the first one being one of the 26 alphabetic characters, and the other six being of the 26 alphabetic or the 10 numeric characters. It is the name of the file to be connected to the unit specified by *un*.

If the FILE parameter is omitted, the file name of files connected for direct or sequential access defaults to FTunF001 on MVS and VM systems. For keyed access files, the name defaults to FTnnK01. The *un* is the integer specified in the UNIT parameter. It must have a leading 0 if *un* is only one digit.

On VSE systems, the default file name is IJSYSun. To take this default, a DLBL statement with the file name must exist.

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 with keys (in which case, the file must be a keyed file). If ACCESS=*acc* is not specified, it is assumed to be SEQUENTIAL.

BLANK=*blk*

blk is a character expression whose value (when any trailing blanks are removed) must be 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 130 and "BZ Format Code" on page 131, respectively. This specifier is only allowed for formatted I/O.

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 connected for formatted or unformatted input/output. If this parameter is omitted and ACCESS=SEQUENTIAL, a value of FORMATTED is assumed; otherwise, a value of UNFORMATTED is assumed.

IOSTAT=*ios*

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

Parameter Used with ACCESS=DIRECT: The following parameter is used only if ACCESS=DIRECT and **must** be specified for such access.

RECL=*rcl*

rcl is an integer expression of length 4. Its value is 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.

Parameters Used with ACCESS=KEYED

IBM Extension

ACTION=*act*

act indicates the kind of processing to be done to a VSAM file. It can be used with any files connected for sequential, direct, or keyed access. It is any character expression whose value can be specified as:

WRITE To open an empty keyed file for the loading of records. The records must be written in ascending key sequence.

READ To open for retrieval a VSAM file that is not empty. Update operations cannot be performed on the file.

READWRITE To open a VSAM file and make retrieval and update operations possible. An update operation is a REWRITE, DELETE, or WRITE statement that causes the replacement, deletion, or addition of a record to a file. Using READWRITE, you can write to an empty keyed file, and you need not write the records in ascending key sequence. READWRITE also enables you to open a VSAM file and then read from it to find out whether or not it is empty.

For sequential or direct access, specify READ or READWRITE.

If the ACTION parameter is omitted, the default for keyed access is READ. The default for sequential or direct access is READWRITE.

The following parameters can be used only if ACCESS=KEYED.

PASSWORD=*pwd*
specifies the password required to access a VSAM file, if the file was password-protected when it was defined with the access method services program. If ACTION=READ, the file's read password is required; otherwise, its update password is required. *pwd* can be any character expression; however, if the character expression exceeds eight characters in length, only the first eight are used.

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 parameter; the missing information for the file is taken from the VSAM catalog. If you will use multiple keys when accessing a keyed file, the KEYS parameter is necessary.

If the file is being loaded (ACTION=WRITE), only the primary key can be specified.

End of IBM Extension

Valid OPEN Statements:

```
OPEN (UNIT=2, IOSTAT=IOS, FILE='DDNAME', STATUS='NEW',
      ACCESS='SEQU'//'ENTIAL   ', FORMAT='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))
```

I/O Unit Is Not Connected to the External File

Successful execution of the OPEN statement connects the I/O unit specified by *un* to the external file specified by *fn* with the parameters specified (or assumed) in the OPEN statement. (See *VS FORTRAN Programming Guide* for the parameters allowed with the various definitions of data sets.)

I/O Unit Is Connected to the External File

A unit connected in any program unit of an executable program is available in any other program unit of the executable program.

The unit reference and the file name are *un* and *fn* in the OPEN statement.

Opening an Already-Open File: If you issue an OPEN statement for a file that is already open and connected to the unit identified in the UNIT parameter, the following occurs:

- The file still exists (*exs* has the value true).
- The unit stays connected to the file.
- The new value of the BLANK specifier comes into effect.
- If the file had the NEW attribute, it is changed to OLD.
- The other attributes remain unchanged.
- The file is not repositioned at the beginning.

If some parameters are specified on the OPEN statement, they must match the attributes of the connection of file (except that BLANK may be different).

Opening a Different File on an Already-Connected Unit: 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 (UNIT=*un*, STATUS=UNKNOWN) followed by an OPEN.

Conditions That Prevent the Execution of OPEN: Any of the following conditions prevent execution of the OPEN statement:

- You specified an invalid unit number, that is, *un*.
- You specified an invalid file name, that is, *fn*.
- You specified invalid values; for example:
  - OLD was specified for a file that does not exist.
  - ACCESS, FORM, REC do not match the actual attributes of an existing file.
  - The RECL=*rcl* value is not positive integer.
  - The OPEN statement specifies a different unit than the one the file is connected to.
  - The KEYS parameter specifies a start:end pair that does not represent a key available for use with the keyed file.

Control transfers to the statement specified in ERR=*stn* or, if ERR=*stn* is not specified, execution of the program is terminated.

PARAMETER Statement

The parameter statement assigns a name to a constant.

Syntax

PARAMETER (*name*1 = *c*1 [, *name*2 = *c*2] ...)

name
: is the name of a specific constant in this program unit (even if it looks like a hexadecimal constant, for example, ZOABC). The name must be defined only once in a PARAMETER statement of a program unit.

c
: is a constant or a constant expression of integer, real, complex, logical, or character type.

Before using the PARAMETER statement, *name* must have been specified by the IMPLICIT statement or an explicit type statement. (Otherwise the predefined conventions are used.)

The type and length of a name of a constant must not be changed by 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 explicitly specified previously or has been been specified as an asterisk, the length is considered to be the length of the value of the character expression (*c*).

If the name (*name*) is of integer, real, or complex type, the corresponding expression (*c*) must be a constant, the name of a constant, or another expression enclosed in parentheses. The exponentiation operator is not permitted unless the exponent is of integer type.

If the name (*name*) is of character type, the corresponding expression (*c*) must be a character expression containing only character constants or names of character constants. The expression result cannot exceed 255 characters in length.

If the name (*name*) is of logical type, the corresponding expression (*c*) must be a logical expression containing only logical constants or names of logical constants.

Each (*name*) is the name of a constant that becomes defined with the value of the expression (*c*) that appears to the right of the equal sign. The value assigned is determined by the rules used for assignment statements (see Figure 18 and Figure 19).

Any name of a constant that appears in an expression (*c*) must be defined by appearing previously on the left of an equal sign in the same or a preceding PARAMETER statement in the same program unit. If it is in the same PARAMETER statement, it must appear to the left of its usage.

Once defined, the name can be used in a subsequent expression or a DATA statement instead of the constant it represents. It must not 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.

IBM Extension

If the name is of integer type of length 2, then the constant value is an integer constant that occupies 2 bytes of storage. Reference to this symbolic name is treated exactly as any reference to an integer variable of length 2 and therefore, in this case, the reference is to an integer constant of length 2. This is the only way an integer constant of this length may be introduced into a source program.

If the name is of logical type of length 1, then the constant value is a logical constant that occupies 1 byte of storage. Reference to this symbolic name is treated exactly as any reference to a logical variable of length 1 and therefore, in this case, the reference is to a logical constant of length 1. This is the only way a logical constant of this length may be introduced into a source program.

End of IBM Extension

PAUSE Statement

The PAUSE statement temporarily halts the execution of the object program and may display a message.

Syntax

PAUSE [*n*]

PAUSE [*'message'*]

n
: a string of 1 through 5 decimal digits.

'message'
: a character constant enclosed in apostrophes and containing alphameric and/or special characters. Within the literal, an apostrophe is indicated by two successive apostrophes.

If either *n* or *'message'* is specified, PAUSE displays the requested information. The program waits until operator intervention causes it to resume execution, starting with the next statement after the PAUSE statement or the next iteration of the DO loop, if it is the last statement of a DO range. For further information, see *VS FORTRAN Programming Guide*.

PRINT Statement

The PRINT statement transfers data from internal storage to an external device.

Syntax

PRINT *fmt* [,*list*]

fmt

can be one of the following:

- A statement number
- An integer variable
- A character constant
- A character array element
- A character array name
- A character expression

IBM Extension

- An array name

End of IBM Extension

- An asterisk that indicates that printing is to be performed according to the data transmission rules of list-directed WRITE

See "WRITE Statement—Formatted with Direct Access" on page 250, for explanations of these format identifiers.

list

is a list of output items and implied DO lists. An output list item can be:

- A variable name
- An array element
- A character substring
- An array name (except the name of an assumed-size array)
- Any expression (except a character expression involving concatenation of operands whose length specification is an asterisk)

See "Implied DO in an Input/Output Statement" on page 92.

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 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.

PRINT *fmt* has the same effect as a WRITE (*un,fmt*) *list*, where *fmt* and *list* are defined as above, and the value of *un* is installation dependent.

Valid PRINT Statement:

```
PRINT*,EIGHT8
```

PROGRAM Statement

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

Syntax

PROGRAM *name*

name
is the name of the main program in which this statement appears.

A main program cannot contain any BLOCK DATA, SUBROUTINE, FUNCTION, or ENTRY statements.

IBM Extension

A RETURN statement may appear; it has the same effect as a STOP statement.

End of IBM Extension

The PROGRAM statement can only be used in a main program but is not required. If it is used, it must be the first statement of the main program. If it is not used, the name of the main program is assumed by this compiler to be MAIN.

The name must not be the same as any other name in the main program or as the name of a subprogram or common block in the same executable program. The name of a program does not have any type and the other specification statements have no effect on this *name*.

Execution of a program begins with the execution of the first executable statement of the main program. A main program may not be referred to from a subprogram or from itself.

READ Statements

The READ statements transfer data from an external device to storage or from an internal file to storage.

Forms of the READ Statement:

IBM Extension

1. "READ Statement—Asynchronous" on page 179

End of IBM Extension

2. "READ Statement—Formatted with Direct Access" on page 182

IBM Extension

3. "READ Statement—Formatted with Keyed Access" on page 186

End of IBM Extension

4. "READ Statement—Formatted with Sequential Access" on page 192

5. "READ Statement—Unformatted with Direct Access" on page 196

IBM Extension

6. "READ Statement—Unformatted with Keyed Access" on page 198

End of IBM Extension

7. "READ Statement—Unformatted with Sequential Access" on page 203

8. "READ Statement—Formatted with Sequential Access to Internal Files" on page 206

9. "READ Statement—List-Directed I/O from External Devices" on page 210

IBM Extension

10. "READ Statement—List-Directed I/O with Internal Files" on page 213

11. "READ Statement—NAMELIST with External Devices" on page 216

12. "READ Statement—NAMELIST with Internal Files" on page 219

End of IBM Extension

IBM Extension

READ Statement—Asynchronous

The asynchronous READ statement transmits unformatted sequential data between direct access or sequential storage devices. The asynchronous READ statement provides high-speed input. The statements are asynchronous because, while data transfer is taking place, other program statements may be executed. An OPEN statement is not permitted for asynchronous I/O. The unit and statement identifier are the only items allowed within the parentheses.

Syntax

READ ([UNIT=]*un*, ID=*id*) [*list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ID=*id*

id is an integer constant or integer expression of length 4. It is the identifier for the READ statement.

list

is an asynchronous I/O list and may have any of four forms:

e
*e*1...*e*2
*e*1...
...*e*2

where:

e

is the name of an array.

***e*1 and *e*2**

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 sequential or direct-access device. The array (*e*) or array elements (*e*1 through *e*2) constitute the receiving area for the data to be read.

The asynchronous READ statement initiates a transmission. The WAIT statement, that must be executed for each asynchronous READ, terminates 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 a special parameter, ID=*id*, is specified within the parentheses of the statement. ID=*id* establishes a unique identification for the READ statement.

Synchronous READ statements may be executed for the file only after all asynchronous READ and WRITE operations have been completed and a REWIND has been executed for the file. Conversely, asynchronous READ statements may be executed for a file previously read synchronously after a REWIND or CLOSE has been executed.

Execution of an asynchronous READ statement initiates reading of the next record on the specified file. The record may contain more or less data than there are bytes in the receiving area. If there is more data, the excess is not transmitted to the receiving area; if there is less, the values of the excess array elements remain unaltered. The extent of the receiving area is determined as follows:

- If *e* is specified, the entire array is the receiving area. In this case, *e* may not be the name of an assumed-size array.
- If *e*1...*e*2 is specified, the receiving area begins at array element *e*1 and includes every element up to and including *e*2. The subscript value of *e*1 must not exceed that of *e*2.
- If *e*1... is specified, the receiving area begins at element *e*1 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 ...*e*2 is specified, the receiving area begins at the first element of the array and includes every element up to and including *e*2.

If *list* is not specified, there is no receiving area, no data is transmitted, and a record is skipped.

Subscripts in the list of the asynchronous READ must not be defined as array elements in the receiving area. If a function reference is used in a subscript, the function reference may not perform I/O on any file.

Given an array with elements of length *len*, transmission begins with the first *len* bytes of the record being placed in the first specified (or implied) array element. Each successive *len* byte of the record is placed in the array element with the next highest subscript value. Transmission ceases after all elements of the receiving area have been filled, or the entire record has been transmitted—whichever occurs first. If the record length is less than the receiving area size, the last array element to receive data may receive fewer than *len* bytes.

The specified array may be multidimensional. Array elements are filled sequentially. Thus, during transmission, the leftmost subscript quantity increases most rapidly, and the rightmost least rapidly.

Any number of program statements may be executed between an asynchronous READ and its corresponding WAIT, subject to the following rules:

- No array element in the receiving area may appear in any such statement. This and the following rules apply also to indirect references to such array elements; that is, reference to or redefinition of any variable or array element associated by COMMON or EQUIVALENCE statements, or argument association with an array element in the receiving area.

- No executable statement may appear that redefines or undefines a variable or array element appearing in the subscript of *e1* or *e2*. See "Valid and Invalid VS FORTRAN Programs" on page 4.

- 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 READ and the corresponding WAIT. Also, no subroutines or functions may be referred to that directly or indirectly refer to the function in the subscript reference, or to which the subscript function directly or indirectly refers.

- No function or subroutine may be executed that performs input or output on the file being manipulated, or that contains object-time dimensions that are in the receiving area (whether they be dummy arguments or in a common block).

Valid READ Statement:

```
READ (ID=10, UNIT=3*IN-3) ACTUAL(3)...ACTUAL(7)
```

End of IBM Extension

READ (Formatted, Direct Access)

READ Statement—Formatted with Direct Access

This READ statement transfers data from an external direct-access device into internal storage. 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 reside on an external file that has been opened for direct access (see "OPEN Statement" on page 168).

Syntax

READ ([UNIT=]*un*, [FMT=]*fmt*, REC=*rec* [, ERR=*stn*]

[, IOSTAT=*ios*]) [*list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

FMT=*fmt*

fmt is a required format identifier and can, optionally, be preceded by FMT=.

If FMT= is not specified, the format identifier must appear second. If both UNIT= and FMT= are specified, all the parameters, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- A statement number
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression

IBM Extension

- An array name

End of IBM Extension

The *statement number* must be the statement number 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 number of a FORMAT statement. The FORMAT statement must be in the same program unit as the READ statement.

The *character constant* must constitute a valid format. The constant must be delimited by apostrophes, must begin with a left parenthesis, and end with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. An apostrophe in a constant enclosed in apostrophes is represented by two consecutive apostrophes.

The *character variable* and *character array element* must contain character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. Blank characters may precede the left parenthesis and character data may follow the right parenthesis. The length of the format identifier must not exceed the length of the array element.

The *character array name* must contain character data whose leftmost characters constitute a valid format identifier. The length of the format identifier may exceed the length of the first element of the array; it is considered the concatenation of all the array elements of the array in the order given by array element ordering.

IBM Extension

The *array name* may be of type integer, real, double precision, logical, or complex.

The data must be a valid format identifier as described under character array name above.

End of IBM Extension

The *character expression* may contain concatenations of character constants, character array elements and character array names. Its value must be a valid format identifier. The operands of the expression must have length specifications that contain only integer constants or names of integer constants. (See Chapter 4, "VS FORTRAN Expressions" on page 35.)

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.

ERR=*stn*
: *stn* is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

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

list
is an I/O list. It can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 92.

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 (un,fmt,REC=rec) list

READ (un,FMT=fmt,REC=rec) list

READ (UNIT=un,FMT=fmt,REC=rec) list

READ (REC=rec,FMT=fmt,UNIT=un)

READ (UNIT=2*IN-10, FMT='(I9)', REC=3)
```

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 OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

This statement permits a programmer 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, a programmer 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. The programmer must specify in the READ statement not only the unit reference number, 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 the type of the records in the direct access file. All the records of a file connected for direct access have the same length.

Data Transmission: A READ statement with FORMAT starts data transmission at the beginning of the record specified by REC=*rec*. The format codes in the format identifier *fmt* are taken one by one and associated with every item of the list in the order they are specified. The number of character data specified by the format code is taken from the record, converted according to the format code and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item of the list or when the end of the record specified by *rec* is reached.

If the list is not specified and the format identifier starts with an I, E, F, D, G, or L format code, or is empty (that is, FORMAT()), the internal record number is increased by one but no data is transferred.

IBM Extension

VS FORTRAN adds that, if the format identifier starts with a Q or Z format code, the internal record number is increased by one but no data is transferred.

End of IBM Extension

Data and I/O List: The length of every FORTRAN record is specified in the RECL of the OPEN statement. If the record *rec* contains **more** data than is necessary to satisfy all the items of the list and the associated format identifier, the remaining data is ignored. If the record *rec* contains **less** data than is necessary to satisfy all the items of the list and the associated format identifier, an error is detected. If the format identifier indicates (for example, slash format code) that data be taken from the next record, then the internal record number *rec* is increased by one and data transmission continues with the next record. The INQUIRE statement can be used to determine the record number.

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Figure 39 on page 334.)

IBM Extension

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 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 parameter 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=*stn*] [, IOSTAT=*ios*]

[, KEY=*key* | , KEYGE=*kge* | , KEYGT=*kgt*]

[, KEYID=*kid*] [, NOTFOUND=*stn*]) [*list*]

Syntax for a Sequential Retrieval Keyed Request

READ ([UNIT=]*un*, [FMT=]*fmt*, [, ERR=*stn*] [, IOSTAT=*ios*]

[, NOTFOUND=*stn* | , END=*stn*]) [*list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

FMT=*fmt*

fmt is a required format identifier and can, optionally, be preceded by FMT=.

If FMT= is not specified, the format identifier must appear second. If both UNIT= and FMT= are specified, all the parameters, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- A statement number
- 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 number* must be the statement number 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 number 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, and must begin with a left parenthesis and must end with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. An apostrophe in a constant enclosed in apostrophes is represented by two consecutive apostrophes.

The *character variable* and *character array element* must contain character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. Blank characters may precede the left parenthesis and character data may follow the right parenthesis. The length of the format identifier must not exceed the length of the array element.

The *character array name* must contain character data whose leftmost characters constitute a valid format identifier. The length of the format identifier may exceed the length of the first element of the array; it is considered the concatenation of all the array elements of the array in the order given by array element ordering.

The *array name* may be of integer, real, double precision, logical, or complex type.

The data must be a valid format identifier as described under character array name above.

The *character expression* may contain concatenations of character constants, character array elements and character array names. Its value must be a valid format identifier. The operands of the expression must have length specifications that contain only integer constants or names of integer constants. (See Chapter 4, "VS FORTRAN Expressions" on page 35.)

ERR=*stn*
stn is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

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

KEY=*key* | KEYGE=*kge* | KEYGT=*kgt*
These parameters cause a record to be retrieved by its key, and the file to be positioned at the end of the record. They supply a full or partial key value which is used as a search argument.

KEY=*key* Specifies that the record to be retrieved is the first record whose key value is identical to the search argument. If the search argument is a partial key (shorter than the keys in the file), the record retrieved is the first one with a key whose leading part is identical to the partial key.

KEYGE=*kge* Specifies the following search criterion for the record to be retrieved: If the file contains a record whose key value is identical to *kge*, the first such record is retrieved. If not, the first record with the next greater key value is retrieved. If *kge* is a partial key (shorter than the keys in the file), the record retrieved is the first one with a key value whose leading part is equal to or greater than the partial key.

KEYGT=*kgt* Specifies that the record to be retrieved is the first one with a key value greater than *kgt*. If *kgt* is a partial key (shorter than the keys in the file), the record retrieved is the first one with a key value whose leading part is greater than the partial key.

key, *kge*, and *kgt* can be a character expression or a data item (a constant, variable, array element, or character substring) of an integer or character type whose length does not exceed the length of the key that is the target of the search. A shorter or partial key is called a generic key.

KEYID=*kid*
kid is an integer expression of length 4. Its value is the relative position of a start-end pair in the list of such pairs in the KEYS parameter of the OPEN statement. For example, KEYID=3 would designate the third start-end pair, and hence the third key, in the KEYS parameter. 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 parameter.

If the KEYID parameter is omitted on the first READ statement for a file opened for keyed access, the first start-end pair on the KEYS parameter is

used. If no KEYS parameter was given on the OPEN statement, KEYID must have a value of 1 or be omitted.

The KEYID parameter can be used only if the KEY, KEYGE, or KEYGT parameter is also used.

NOTFOUND=*stn*
stn is the number of an executable statement that is given control when a record-not-found condition occurs. See "Record Not Found" below for an explanation of this condition.

END=*stn*
stn is the number 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 *stn*.

This parameter 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 92.

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 parameter 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 parameter. For a sequential retrieval keyed request, data transmission begins at the beginning of the record at which the file is currently positioned. The format codes in the format identifier *fmt* are taken one by one and associated with every item in the list in the order they are specified. The number and character data specified by the format code is taken from the record, converted according to the format code, and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item in the list or when the end of the record has been reached.

Data and I/O List: If the record contains more data than is necessary to satisfy all the items of the list and the associated format specification, the extra data is skipped over. The next sequential retrieval READ statement will start with the next sequential record. (This is the record with the next higher key value if the key value is unique or the next record with the same key if the key value is not unique.) If the record contains *less* data than is necessary to satisfy all the items of the list and the associated format identifier, an error is detected.

If the list is not specified and the format identifier starts with an I, E, F, D, G, or L format code or is empty (that is, FORMAT()), a record is skipped over.

VS FORTRAN adds the Q and Z format codes to the list.

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, then 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 Figure 39 on page 334.)

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 parameter on the sequential retrieval keyed request is treated as an END parameter 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 parameter. In the absence of the NOTFOUND parameter, 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.

End of IBM Extension

READ Statement—Formatted with Sequential Access

This READ statement transfers data from an external I/O device to storage. 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 reside in an external file that is connected for sequential access to a unit. (See "OPEN Statement" on page 168.)

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 ([UNIT=]*un*, [FMT=]*fmt* [, ERR=*stn*] [, END=*stn*]

[, IOSTAT=*ios*]) [*list*]

READ *fmt* [, *list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is one of the following:

- An integer expression of length 4 whose value must be zero or positive
- An asterisk (*) representing an installation-dependent unit

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

In the form of the READ in which *un* is not specified, *un* is installation dependent.

FMT=*fmt*

fmt is a required format identifier. It can optionally be preceded by FMT=.

If FMT= is not specified, the format identifier must appear second. If both UNIT= and FMT= are specified, all the parameters, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- A statement number
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression

IBM Extension

- An array name

End of IBM Extension

See "READ Statement—Formatted with Direct Access" on page 182 for explanations of these format identifiers.

ERR=*stn*
: *stn* is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

END=*stn*
: *stn* is the number 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 *stn*.

IOSTAT=*ios*
: *ios* is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is detected, and zero if no error is detected. VSAM return and reason codes are placed in *ios*.

list
: is an I/O list. It can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 92. 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 (un,fmt) list

READ (un, FMT=fmt) list

READ (UNIT=un, FMT=fmt) list   FMT=fmt can appear first.

READ fmt, list

READ (5,98) A,B,(C(I,K),I=1,10)

READ (IOSTAT=IOS, UNIT=2*IN-10, FMT='(I9)', END=3600)

READ (10,22) AA,BB,CC
```

Invalid READ Statements:

`READ (fmt,un)`	un must appear before fmt.
`READ (FMT=fmt, un) list`	un must appear first because UNIT= is not specified.
`READ (fmt, UNIT=un) list`	FMT= must be used because UNIT= is specified.
`READ FMT=fmt, list`	FMT= must not be used in this form of READ.

If this READ statement is encountered, the unit specified must exist and the file must be connected 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 FORMAT starts data transmission at the beginning of a record. The format codes in the format identifier *fmt* are taken one by one and associated with every item of the list in the order they are specified. The number of character data specified by the format code is taken from the record, converted according to the format code, and transmitted into the storage associated with the corresponding item in the list. Data transmission stops when data has been transmitted to every item of the list, or when the end of file is reached.

Data and I/O List: If the record contains more data than is necessary to satisfy all the items of the list and the associated format specification, the extra data is skipped over. The next READ statement with FORMAT will start with the next record if no other I/O statement is executed on that file. If the record contains less data than is necessary to satisfy all the items of the list and the associated format identifier, see "End of File" below.

If the list is not specified and the format identifier starts with an I, E, F, D, G, or L format code or is empty (that is, FORMAT()), a record is skipped over.

IBM Extension

VS FORTRAN adds the Q and Z format codes to the list.

End of IBM Extension

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor 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 Figure 39 on page 334.)

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, object program execution is terminated when the end of the file is encountered.

READ Statement—Unformatted with Direct 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 direct access. (See "OPEN Statement" on page 168.)

Syntax

READ ([UNIT=]*un*, REC=*rec* [, ERR=*stn*] [, IOSTAT=*ios*]

[, NUM=*n*]) [*list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters 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.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error is detected. VSAM return and reason codes are placed in *ios*.

IBM Extension

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 parameter suppresses the indication of an error that would occur if the number of data bytes represented by the I/O list is greater than the number of bytes in the record. In this case, *n* is set to a value which is the number of bytes in the record. Data from subsequent records is not read into the remaining I/O list items.

End of IBM Extension

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 92.

Valid READ Statements:

```
READ (un,REC=rec) list

READ (REC=rec, UNIT=un)

READ (IOSTAT=IOS, UNIT=11, REC=3) ACTUAL(3)(1:)
```

If this READ statement is encountered, the unit 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 OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

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 FORTRAN 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 parameter is given.

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Figure 39 on page 334.)

IBM Extension

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 parameter 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=*stn*] [, IOSTAT=*ios*]

{, KEY=*key* | , KEYGE=*kge* | , KEYGT=*kgt* }[, KEYID=*kid*]

[, NOTFOUND=*stn*] [, NUM=*n*]) [*list*]

Syntax for a Sequential Retrieval Keyed Request

READ ([UNIT=]*un*, [, ERR=*stn*] [, IOSTAT=*ios*]

[, NOTFOUND=*stn* | , END=*stn*] [, NUM=*n*]) [*list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error is detected. VSAM return and reason codes are placed in *ios*.

KEY=*key* | KEYGE=*kge* | KEYGT=*kgt*

These parameters 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 parameter of the OPEN statement. For example, KEYID=3 would designate the third start-end pair, and hence the third key, in the KEYS parameter. 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 parameter.

If the KEYID parameter is omitted on the first READ statement for a file opened for keyed access, the first start-end pair on the KEYS parameter is used. If no KEYS parameter was given on the OPEN statement, KEYID must have a value of 1 or be omitted.

The KEYID parameter can be used only if the KEY, KEYGE, or KEYGT parameter is also used.

NOTFOUND=*stn*
stn is the number of an executable statement that is given control when a record-not-found condition occurs. See "Record Not Found" below for an explanation of this condition.

END=*stn*
stn is the number 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 *stn*.

This parameter 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 parameter suppresses the indication of an error that would occur if the number of data bytes represented by the I/O list is greater than the number of bytes in the record. In this case, *n* is set to a value which is the number of bytes in the record. Data from subsequent records is not read into the remaining I/O list items.

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 92.

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 parameter 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 parameter. 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 parameter was given.

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Figure 39 on page 334.)

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 parameter on the sequential retrieval keyed request is treated as an END parameter 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 parameter. In the absence of the NOTFOUND parameter, 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.

End of IBM Extension

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. (See "OPEN Statement" on page 168.)

The sequential I/O statements without format control process records one after the other from the beginning of an external file to its end.

The ENDFILE, REWIND, and BACKSPACE statements may be used to manipulate the file.

Syntax

READ ([UNIT=]*un* [, ERR=*stn*] [, END=*stn*] [, NUM=*n*]

[, IOSTAT=*ios*]) [*list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

END=*stn*

is the number 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 *stn*.

IBM Extension

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 parameter suppresses the indication of an error that would occur if the number of data bytes represented by the I/O list is greater than the number of bytes in the record. In this case, *n* is set to a value which is the number of bytes in the record. Data from subsequent records is not read into the remaining I/O list items.

End of IBM Extension

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected. VSAM return and reason codes are placed in *ios*.

list

is an I/O list. It can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 92.

Valid READ Statements:

```
READ (un) list

READ (UNIT=un) list

READ (un)

READ (IOSTAT=IOS, UNIT=11)

READ (12) DD,EE,FF
```

Invalid READ Statements:

```
READ un, list          un must be in parentheses (and a comma
                       must not be specified).

READ, list             (un) must be specified (and a comma
                       must not be specfied).
```

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 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, see "End of File" below.

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor 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 Figure 39 on page 334.)

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.

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. The user specifies in a FORMAT statement (or in a reference to a FORMAT statement) 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=*stn*] [, END=*stn*]

[, IOSTAT=*ios*]) [*list*]

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It can be the name of:

- A character variable
- A character array
- A character array element
- A character substring

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

FMT=*fmt*

fmt is a required format identifier. It can, optionally, be preceded by FMT=.

The format identifier can be:

- A statement number
- An integer variable
- A character constant
- A character variable
- A character array name
- A character array element
- A character expression

IBM Extension

- An array name

End of IBM Extension

See "READ Statement—Formatted with Direct Access" on page 182 for explanations of these format identifiers.

The format specification must **not** be:

- In the area *un*
- Associated with *un* through EQUIVALENCE, COMMON or argument passing

If FMT= is not specified, the format specification must appear second. If both UNIT= and FMT= are specified, all the parameters, except *list*, can appear in any order.

ERR=*stn*
: *stn* is the number of an executable statement in the same program unit as the READ statement. If an error is detected, transfer is made to *stn*.

END=*stn*
: *stn* is the number 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 *stn*.

IOSTAT=*ios*
: *ios* is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected. VSAM return and reason codes are placed in *ios*.

list
: is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 92.

An item in the list must **not** be:

- Contained in the area represented by *un*
- Associated with any part of *un* through EQUIVALENCE, COMMON, or argument passing

READ (Formatted, Sequential Access, Internal)

Valid READ Statements:

```
READ (un,fmt) list

READ (un,FMT=fmt) list

READ (UNIT=un,FMT=fmt) list
```

Invalid READ Statements:

```
READ (fmt,un) list            un must appear before fmt.

READ (FMT=fmt,un) list        un must appear first because
                              UNIT= is not specified.

READ (fmt,UNIT=un) list       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, character array element specified by *un* when the READ statement is executed.

If a record contains *more* data than is necessary to satisfy all the items in the list and the associated format identifier, the remaining data is ignored.

If a record contains **less** data than is necessary to satisfy all the items in the list and the associated format identifier, an error is detected.

If the format identifier (for example, slash format code) indicates that further data is needed beyond the data contained in the character variable, character substring, or the last array element of a character array, an end of file is detected. If it is not the last array element in the character array, data is taken from the next array element.

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Figure 39 on page 334.)

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, then the format numbered 300 replaces the format numbered 200 and is used for processing the data. This is done by assigning 300 to the integer variable J. Statement 5 reads the file and performs the conversion, using the appropriate FORMAT statement and assigning values to A1, A2, and A3.

READ (List-Directed, External)

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

This statement transfers data from an external device into internal storage. The type of the items specified in this statement determines the conversion to be performed. The data resides on an external file that is connected for sequential access to a unit (see "OPEN Statement" on page 168).

Syntax

READ ([UNIT=]*un*, [FMT=]* [, ERR=*stn*] [, END=*stn*]

[, IOSTAT=*ios*]) [*list*]

READ * [, *list*]

UNIT=*un*
un is the reference to the number of an I/O unit. *un* is one of the following:

- An integer expression of length 4 whose value must be zero or positive
- An asterisk (*) representing an installation-dependent unit

un is required in the first form of the READ statement and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

In the form of the READ in which *un* is not specified, *un* is installation dependent.

FMT=*
specifies that a list-directed READ is to be executed. It can optionally be preceded by FMT=.

If FMT= is not specified, the format identifier must appear second. If both UNIT= and FMT= are specified, all the parameters, except *list*, can appear in any order.

ERR=*stn*
stn is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

END=*stn*
stn is the number 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 *stn*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected. VSAM return and reason codes are placed in *ios*.

list

is an I/O list and can contain variable names, array element names, character substring names, array names (except names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 92.

Valid READ Statements:

```
READ (un,*) list
READ (un,FMT=*) list
READ (FMT=*,UNIT=un) list
READ (*,*) list
READ *, list
READ (IOSTAT=IACT(1), UNIT=3*IN-2, FMT=*) ACTUAL(1)
```

Invalid READ Statements:

```
READ (*,un) list           un must appear before *.

READ (FMT=*,un) list       un must appear first because
                           UNIT= is not specified.

READ (*,UNIT=un) list      FMT= must be used because
                           UNIT= is specified.

READ FMT=*, list           FMT= must not be specified in the
                           second form of syntax.
```

If this READ statement is encountered, the unit specified by *un* must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language, and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

Data Transmission: A READ statement with list-directed I/O accessing an external file starts data transmission at the beginning of a record. One value on the external file is transferred to each item of the list in the order they are specified. The conversion to be performed depends on the type and length of the name of the item in the list. Data transmission stops when data has been transmitted to every item in the list, when a slash separator is encountered in the file or when the end of file is reached.

Data and I/O List: If the record contains more data than is necessary to satisfy all the items of the list, the extra data is skipped over. The next READ statement with list-directed I/O will start with the next record if no other I/O statement is executed on that file. If the record contains less data than is necessary to satisfy the list, and the record does not have a slash after the last element, an error is detected. If the list has not been satisfied when a slash separator is found, the remaining items in the list remain unaltered and execution of the READ is terminated.

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Figure 39 on page 334.)

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, object program execution is terminated when the end of the file is encountered.

IBM Extension

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=*stn*] [, END=*stn*]

[, IOSTAT=*ios*]) [*list*]

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It can be the name of:

- A character variable
- A character array
- A character array element
- A character substring

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

FMT=*

* specifies that a list-directed READ is to be executed. It can optionally be preceded by FMT=.

If FMT= is not specified, * must appear second. If both UNIT= and FMT= are specified, all the parameters, except *list*, can appear in any order.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

END=*stn*

stn is the number 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 *stn*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected.

list
is an I/O list and can contain variable names, array element names, character substring names, array names (except names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 92.

Valid READ Statements:

```
READ (un,*) list
READ (un,FMT=*) list
READ (FMT=*,UNIT=un) list
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.

End of IBM Extension

IBM Extension

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 ([UNIT=]*un*, [FMT=]*name* [, ERR=*stn*]

[, END=*stn*] [, IOSTAT=*ios*])

READ *name*

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is one of the following:

- An integer expression of length 4 whose value must be zero or positive
- An asterisk (*) representing an installation-dependent unit

un is required in the first form of the READ statement and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

In the form of the READ in which *un* is not specified, *un* is installation dependent.

FMT=*name*

name is a NAMELIST name. See "NAMELIST Statement" on page 166.

If FMT= is not specified, the NAMELIST name must appear second. If both UNIT= and FMT= are specified, all the parameters can appear in any order.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

END=*stn*

stn is the number 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 *stn*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected. VSAM return and reason codes are placed in *ios*.

Valid READ Statements:

```
READ (un,name)

READ (IN+IN+3, NAMEIN, IOSTAT=IOS)
```

Invalid READ Statements:

```
READ (name,un)          un must appear before name.

READ (un,name) list     list must not be specified.
```

If this READ statement 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 FORTRAN program with the data itself. There is no format identifier but the data is converted according to the type of data in the FORTRAN program. The data on the external file must be in a specific format. See "NAMELIST Input Data" on page 167.

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 67 and "REWIND Statement" on page 225).

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. (See "NAMELIST Input Data" on page 167 for the format of the input data.)

Control is transferred to the statement specified by ERR if a transmission error is detected. No indication is given of which record or records could not be read, only that the error occurred during transmission of data. If IOSTAT is specified, a

positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines may be used to detect and handle these errors. (See Figure 39 on page 334.)

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, object program execution is terminated when the end of the file is encountered.

End of IBM Extension

IBM Extension

READ Statement—NAMELIST with Internal Files

This statement transfers data from one area of internal storage to one or more other areas of internal storage. The area of internal storage that is read from is called an internal file. The type of the items specified in an associated NAMELIST list determines the conversions to be performed.

Syntax

READ ([UNIT=]*un*, [FMT=]*name* [, ERR=*stn*] [, END=*stn*]

[, IOSTAT=*ios*])

UNIT=*un*
un is the reference to an area of internal storage called an internal file. It can be the name of:

- A character variable
- A character array
- A character array element
- A character substring

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

FMT=*name*
name is a NAMELIST name. See "NAMELIST Statement" on page 166.

If FMT= is not specified, the NAMELIST name must appear second. If both UNIT= and FMT= are specified, all the parameters can appear in any order.

ERR=*stn*
stn is the number of an executable statement in the same program unit as the READ statement. If an error is detected, control is transferred to *stn*.

END=*stn*
stn is the number 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 *stn*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected.

Valid READ Statements:

```
READ (un,name)

READ (CHARVR, NAMEIN, IOSTAT=IOS)
```

The NAMELIST I/O statements associate the name given to the data in the FORTRAN program with the data itself. There is no format identifier, but the data is converted according to the type of data in the FORTRAN program. The data in the internal file must be in a specific format. See "NAMELIST Input Data" on page 167.

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. (See "NAMELIST Input Data" on page 167 for the format of the input data.)

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
&NL 1 A=5,C=10,B=6,&END
```

Then A is assigned the value 5, B the value 6, and C the value 10.

End of IBM Extension

REAL Type Statement

See "Explicit Type Statement" on page 103.

RETURN Statement

The RETURN statement returns control to a calling program.

IBM Extension

In a main program, a RETURN statement performs the same function as a STOP statement.

End of IBM Extension

The RETURN statement can be used in either a **function** or a **subroutine** subprogram. A RETURN statement cannot terminate the range of a DO-loop.

RETURN Statement in a Function Subprogram

Function subprograms may contain RETURN statements. The RETURN statement signifies a logical conclusion of the computation and returns the computed function value and control to the calling program. (See "FUNCTION Statement" on page 137.)

Syntax

```
RETURN
```

Execution of a RETURN statement terminates the association between the dummy arguments of the subprogram and the current actual arguments. All entities (that is, common blocks, variables, or arrays) within the subprogram become undefined except:

- Entities specified in SAVE statements (see "SAVE Statement" on page 232)
- Entities given an initial value in a DATA or explicit specification statement and whose initial values were not changed
- Entities in a blank common block
- Entities in a named common block that appear in the subprogram and appear in at least one other program unit that is referring either directly or indirectly to the subprogram

All variables that are defined with a statement number become undefined regardless of whether the variable is in a common block or specified in a SAVE statement.

A function subprogram must not be referred to twice during the execution of an executable program without the execution of a RETURN statement in that subprogram. (See "END Statement" on page 94.)

RETURN Statement in a Subroutine Subprogram

Subroutine subprograms may contain RETURN statements. The RETURN statement signifies a logical conclusion of the computation and returns control to the calling program. (See "SUBROUTINE Statement" on page 238.)

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.

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 232)
- Entities given an initial value in a DATA or explicit specification statement and where initial values were not changed
- Entities in a blank common block
- Entities in a named common block that appear in the subprogram and appear in at least one other program unit that is referring either directly or indirectly to the subprogram.

All variables that are defined with a statement number become undefined regardless of whether the variable is in a common block or specified in a SAVE statement.

A subprogram must not be referred to twice during the execution of an executable program without the execution of a RETURN statement in that subprogram.

A CALL statement that is used with a RETURN *m* form may be best understood by comparing it to a CALL and computed GO TO statement in sequence. For example, the following CALL statement:

```
CALL SUB (P,*20,Q,*35,R,*22)
```

is equivalent to:

```
CALL SUB (P,Q,R,I)
GO TO (20,35,22),I
```

where the index I is assigned a value of 1, 2, or 3 in the called subprogram.

REWIND Statement

The REWIND statement repositions a sequentially accessed file at the beginning of the first record of the file. The external file must be connected when you execute the statement. (See "OPEN Statement" on page 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=*err*] [, IOSTAT=*ios*])

UNIT=*un*
: *un* is the reference to the number of an I/O unit. *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 not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*err*
: is optional. *err* is a statement number. If an error occurs in the execution of the REWIND statement, control is transferred to the statement labeled *err*. That statement must be executable and must be in the same program unit as the REWIND statement. If ERR=*err* is omitted, execution halts when an error is detected.

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

If UNIT= is specified, all the parameters can appear in any order; otherwise, *un* must appear first.

If the unit specified by *un* is connected, it must be connected for sequential or keyed access. If it is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN specifying sequential access is performed to a default file name. If the file is not preconnected, an error is detected.

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*.

IBM Extension

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.

End of IBM Extension

Transfer is made to the statement specified by the ERR parameter 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 parameter, if present, or with the next statement if ERR is not specified. If the ERR parameter and the IOSTAT parameter 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)
```

IBM Extension

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 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*, [FMT=]*fmt* [, ERR=*stn*] [, IOSTAT=*ios*]

[, DUPKEY=*stn*] *list*

UNIT=*un*

un is a reference to the number of an I/O unit. *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 not specified, *un* must appear first in the statement. The other parameters can appear in any order. If UNIT= is specified, all the parameters 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 not specified, the format identifier must appear second. If both UNIT= and FMT= are specified, all the parameters, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- A statement number
- An integer variable
- A character constant
- A character variable
- A character array element
- An array name
- A character expression

See "WRITE Statement—Formatted with Direct Access" on page 250 for explanations of these format identifiers.

ERR=*stn*
: *stn* is the number of an executable statement in the same program unit as the REWRITE statement. If an error is detected, control is transferred to *stn*.

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

DUPKEY=*stn*
: *stn* 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.

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 of 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 92. 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
```

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: 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 the list is not specified and the format specification starts with an I, E, F, D, G, or L, or is empty (that is, FORMAT( )), a blank record is written. A blank record is also written when the format specification starts with a Q or Z format code.

Control is transferred to the statement specified by ERR if a transmission error is detected. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines can be used to detect and handle these errors. (See Figure 39 on page 334.)

End of IBM Extension

IBM Extension

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=*stn*] [, IOSTAT=*ios*]

[, DUPKEY=*stn*][,NUM=*n*]) *list*

UNIT=*un*
un is a reference to the number of an I/O unit. *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 not specified, *un* must appear first in the statement. The other parameters can appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*stn*
stn is the number of an executable statement in the same program unit as the REWRITE statement. If an error is detected, control is transferred to *stn*.

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

DUPKEY=*stn*
stn 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 parameter 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. 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 of 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 92. 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.

Control is transferred to the statement specified by ERR if a transmission error is detected. If IOSTAT is specified, a positive integer value is assigned to *ios* when the error is detected. If ERR is not specified, execution continues with the next statement.

Errors caused by the length of the data record or the value of the data are not considered transmission errors. These errors do not cause IOSTAT to be set positive nor will transfer be made to the statement specified by ERR. The extended error handling subroutines can be used to detect and handle these errors. (See Figure 39 on page 334.)

End of IBM Extension

SAVE Statement

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

Because VS FORTRAN saves these names without user action, the SAVE statement serves only as a documentation aid.

Syntax

SAVE [*name*1 [, *name*2] ...]

name
: is a named common block name preceded and followed by a slash, a variable name, or an array name. Redundant appearances of an item are not permitted.

Dummy argument names, procedure names, and names of entities in a common block must not appear in a SAVE statement.

A SAVE statement without a list is treated as though it contained the names of all common items in that program unit.

The appearance of a named common block in a SAVE statement has the effect of specifying all entities in that named common block.

The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

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

Within a function or subroutine subprogram, an entity (that is, a common block, variable, or array) specified by a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram.

If a local entity that is specified by a SAVE statement and is not in a common block is in a defined state at the time a RETURN or END statement is executed in a subprogram, that entity is defined with the same value at the next reference of that subprogram. An entity in a common block never becomes undefined as a result of the execution of a RETURN or END statement in a program unit that

does not reference that common block. The entities in a named common block may become undefined or redefined by some other program unit.

Statement Function Statement

A statement function definition specifies operations to be performed whenever that statement function name appears as a function reference in another statement in the same program unit.

Syntax

name [([*arg*1 [, *arg*2] ...])] = *m*

name
is the statement function name (see "Names" on page 11).

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. Parentheses must be specified even if no dummy argument is specified.

m
is any arithmetic, logical, or character expression. Any statement function appearing in this expression must have been defined previously. In a function or subroutine subprogram, this expression can contain dummy arguments that appear in the FUNCTION, SUBROUTINE, or ENTRY statements of the same program unit. (See Chapter 4, "VS FORTRAN Expressions" on page 35, for evaluation and restrictions of this expression.)

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 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 dummy arguments enclosed in parentheses following the function name are dummy variables for which the arguments given in the function reference are substituted when the function reference is encountered. The same dummy arguments may be used in more than one statement function definition, and may be used as variables of the same type outside the statement function definitions, including dummy arguments of subprograms. The length specification of a dummy

argument of type character must be an arithmetic expression containing only integer constants or names of integer constants.

An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument. It cannot be a character expression involving concatenation of one or more operands whose length specification is an asterisk.

If an actual argument is of character type, the associated dummy argument must be of character type and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument is greater than the length of an associated dummy argument, the leftmost characters of the actual argument are associated with the dummy argument.

The name of a statement function must not appear in an EXTERNAL statement and must not be used as an actual argument.

For example, The statement:

```
FUNC(A,B) = 3.*A+B**2.+X+Y+Z
```

defines the statement function FUNC, where FUNC is the function name and A and B are the dummy arguments. The expression to the right of the equal sign defines the operations to be performed when the function reference appears in an arithmetic statement.

The function reference might appear in a statement as follows:

```
C = FUNC(D,E)
```

This is equivalent to:

```
C = 3.*D+E**2.+X+Y+Z
```

Notice the correspondence between the dummy arguments A and B in the function definition and the actual arguments D and E in the function reference.

Valid Statement Function Definitions and References:

Definition	***Reference***
`SUM(A,B,C,D) = A+B+C+D`	`NET = GROS-SUM(TAX,COVER,HOSP, STOC)`
`FUNC(Z) = A+X*Y*Z`	`ANS = FUNC(RESULT)`
`VALID(A,B) = .NOT. A .OR. B`	`VAL = TEST .OR. VALID(D,E)`
	`BIGSUM = SUM(A,B,SUM(C,D,E,F),G(I))`

Invalid Statement Function Definitions:

SUBPRG(3,J,K)=3*I+J**3	Arguments must be variables.
SOMEF(A(I),B)=A(I)/B+3.	Arguments must not be array elements.
SUBPROGRAM(A,B)=A**2+B**2	Function name exceeds limit of six characters.
3FUNC(D)=3.14*E	Function name must begin with an alphabetic character.
BAD(A,B)=A+B+BAD(C,D)	A recursive definition is not permitted.
NOGOOD(A,A)=A*A	Arguments are not distinct variable names.

Invalid Statement Function References:

(The functions are defined as above.)

WRONG = SUM(TAX,COVER)	Number of arguments does not agree with above definition.
MIX = FUNC(I)	Type of argument does not agree with above definition.

Statement Numbers

Statement numbers identify statements in VS FORTRAN programs. Any statement can have a number, and may be written in either fixed form or free form. See "Source Language Statements" on page 7.

Fixed Form Statement Numbers

Fixed form statement numbers have the following attributes:

- They contain 1 to 5 decimal digits (not zero) and are on a noncontinued line.
- Blanks and leading zeros are ignored.
- They are in columns 1 through 5.

IBM Extension

Free-Form Statement Numbers

Free-form statement numbers 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 number and the first nonblank character following.

End of IBM Extension

See "ASSIGN Statement" on page 59.

STOP Statement

The STOP statement terminates the execution of the object program and may display a message.

Syntax

STOP [*n*]

STOP [*'message'*]

n

a string of 1 through 5 decimal digits.

'message'

a character constant enclosed in apostrophes and containing alphameric and/or special characters. Within the literal, an apostrophe is indicated by two successive apostrophes.

If either *n* or *'message'* is specified, STOP displays the requested information. For further information, see *VS FORTRAN Programming Guide*.

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

SUBROUTINE Statement

The SUBROUTINE statement identifies a subroutine subprogram.

Syntax

SUBROUTINE *name* [([*arg*1] [,*arg*2] ...])]

name

is the subroutine name. (See "Names" on page 11.)

arg

is a distinct dummy argument (that is, it may appear only once within the statement). There need not be any arguments, in which case the parentheses may be omitted. Each argument used must be a variable or array name, the dummy name of another subroutine or function subprogram, or an asterisk, where the character * denotes a return point specified by a statement number in the calling program.

Because the subroutine is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

The SUBROUTINE statement must be the first statement in the subprogram. The subroutine subprogram may contain any FORTRAN statement except a FUNCTION statement, another SUBROUTINE statement, a BLOCK DATA statement, or a PROGRAM statement. If an IMPLICIT statement is used in a subroutine subprogram, it must follow the SUBROUTINE statement and may only be preceded by another IMPLICIT statement, a PARAMETER, FORMAT, or ENTRY statement.

The subroutine name must not appear in any other statement in the subroutine subprogram. It must not be the same as any name in the program unit or as the PROGRAM name, a subroutine name, or a common block name in any other program unit of the executable program. The subroutine subprogram may use one or more of its arguments to return values to the calling program. An argument so used will appear on the left side of an arithmetic, logical, or character assignment statement, in the list of a READ statement within the subprogram, or as an argument in a CALL statement or function reference that is assigned a value by the subroutine or function referred to.

The dummy arguments (*arg*1, *arg*2, *arg*3,..., *arg*n) 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 30.

The subroutine subprogram can be a set of commonly used computations, but it need not return any results to the calling program. For information about using

RETURN and END statements in a subroutine subprogram, see "END Statement" on page 94 and "RETURN Statement" on page 222.

Actual Arguments in a Subroutine Subprogram

The actual arguments in a subroutine reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. The use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.

If an actual argument is of character type, the associated dummy argument must be of character type and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument is greater than the length of an associated dummy argument, the leftmost characters of the actual argument are associated with the dummy argument.

An actual argument in a subroutine reference must be one of the following:

- An expression, except for a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses (unless the operand is the name of a constant)
- An array name
- An intrinsic function name
- An external procedure name
- A dummy procedure name
- An alternate return specifier (statement number preceded by an asterisk)

An actual argument in a subroutine reference may be a dummy argument name that appears in a dummy argument list within the subprogram containing the reference. An asterisk dummy argument cannot be used as an actual argument in a subprogram reference.

Dummy Arguments in a Subroutine Subprogram

The dummy arguments of a subprogram appear after the subroutine name and are enclosed in parentheses. They are replaced at the time of execution of the CALL statement by the actual arguments supplied in the CALL statement in the calling program.

Dummy arguments must follow certain rules:

- None of the dummy argument names may appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, SAVE, INTRINSIC, or NAMELIST statement except as common block names.
- A dummy argument name must not be the same as the entry point name appearing in a PROGRAM, FUNCTION, SUBROUTINE, ENTRY, or statement function definition in the same program unit.

- The dummy arguments must correspond in number, order, and type to the actual arguments.

- If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, an array element, a substring, or an array. A constant, name of constant, subprogram name, or expression should not be written as an actual argument unless the programmer is certain that the corresponding dummy argument is not assigned a value in the subprogram.

- A referenced subprogram cannot assign new values to dummy arguments that are associated with other dummy arguments within the subprogram or with variables in common.

- The subprogram reserves no storage for the dummy argument, using the corresponding actual argument in the calling program for its calculations. Thus the value of the actual argument changes as soon as the dummy argument changes.

Valid Examples:

1. Definition of subroutines SUB1 and SUB2: The following illustrates the two ways to define a subroutine without any 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 WITH A VARIABLE AND TWO ARRAY NAMES

CALL SUB3(Y, W, X)

  CALL SUB3 WITH AN ARRAY ELEMENT AND TWO ARRAY NAMES

CALL SUB3(Z(3), X, W)

  CALL SUB3 WITH A CONSTANT AND TWO ARRAY NAMES

CALL SUB3(2.5, W, X)

  CALL SUB3 WITH AN EXPRESSION AND TWO ARRAY NAMES

CALL SUB3(5*Y, X, W)
```

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 USING A LOGICAL VARIABLE

CALL SUB4(L)

  CALL USING A LOGICAL CONSTANT

CALL SUB4(.FALSE.)

  CALL USING A LOGICAL EXPRESSION

CALL SUB4(X(5) .EQ. Y)
```

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 USING A CHARACTER VARIABLE

CALL SUB5(C1)

  CALL USING A CHARACTER EXPRESSION

CALL SUB5(C1 // C2)
```

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.

```
  CALL PASSING 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.

```
  CALL PASSING 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)
```

IBM Extension

TRACE OFF Statement

The TRACE OFF statement stops the display of program flow by statement number.

Syntax

```
TRACE OFF
```

TRACE OFF may appear anywhere within a debug packet. After a TRACE ON statement, tracing continues until a TRACE OFF statement is encountered.

TRACE ON Statement

The TRACE ON statement initiates the display of program flow by statement number.

Syntax

```
TRACE ON
```

TRACE ON is executed only when the TRACE option appears in a DEBUG packet. (See "DEBUG Statement" on page 82.) 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 numbers in that program are not traced.

Each time a statement with an external statement number is executed, a record of the statement number is made on the debug output file.

For a given debug packet, the TRACE ON statement takes effect immediately before the execution of the statement specified in the AT statement.

End of IBM Extension

Unconditional GO TO

See "GO TO Statements" on page 142.

IBM Extension

WAIT Statement

The WAIT statement completes the data transmission begun by the corresponding asynchronous READ or WRITE statement.

Syntax

WAIT ([UNIT=]*un*, *plist*) [*list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an unsigned integer expression of length 4.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

plist

is a parameter list that contains (in any order) one or more of the following forms:

ID=*id*

is required. *id* is an integer constant or integer expression of length 4.

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 COMMON or EQUIVALENCE statement, or argument association with an array element in the receiving area.
- If a function reference appears in the subscript expression of *e*1 or *e*2, the function may not be referred to in the expression *id*. Also, no functions or subroutines may be referred to by the expression that directly or indirectly refers to the subscript function, or to which the subscript function directly or indirectly refers.

COND=*i*1

is optional. *i*1 is an integer variable name of length 4.

If COND=*i*1 is specified, the variable *i*1 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*
is optional. *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 parameter must be specified. If the WAIT is completing an asynchronous WRITE, *i2* remains unaltered.

list
is optional. It is an asynchronous I/O list as specified for the asynchronous READ and WRITE statements.

If a list is included, it must specify the same receiving or transmitting area as the corresponding asynchronous READ or WRITE statement. It must not be specified if the asynchronous READ did not specify a list.

WAIT redefines a receiving area and makes it available for reference, or makes a transmitting area available for redefinition.

The corresponding asynchronous READ or WRITE, which need not appear in the same program unit as the WAIT, is the statement that:

- Was not completed by the execution of another WAIT
- Refers to the same file as the WAIT
- Contains the same value for *id* in the ID=*id* form as did the asynchronous READ or WRITE when it was executed

The correspondence between WAIT and an asynchronous READ or WRITE holds for a particular execution of the statements. Different executions may establish different correspondences.

When the WAIT is completing an asynchronous READ, the subscripts in the list may not refer to array elements in the receiving area. If a function reference is used in a subscript, the function reference may not perform I/O on any file.

Valid WAIT Statements:

```
WAIT (8,ID=1) ARRAY(101)...ARRAY(500)

WAIT (9,ID=1,COND=ITEST)

WAIT (8,ID=1,NUM=N)

WAIT (9,ID=2)
```

End of IBM Extension

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:

IBM Extension

1. "WRITE Statement—Asynchronous" on page 247

End of IBM Extension

2. "WRITE Statement—Formatted with Direct Access" on page 250

IBM Extension

3. "WRITE Statement—Formatted with Keyed Access" on page 254

End of IBM Extension

4. "WRITE Statement—Formatted with Sequential Access" on page 258
5. "WRITE Statement—Unformatted with Direct Access" on page 262

IBM Extension

6. "WRITE Statement—Unformatted with Keyed Access" on page 265

End of IBM Extension

7. "WRITE Statement—Unformatted with Sequential Access" on page 268
8. "WRITE Statement—Formatted with Sequential Access to Internal Files" on page 270
9. "WRITE Statement—List-Directed I/O to External Devices" on page 274

IBM Extension

10. "WRITE Statement—List-Directed I/O with Internal Files" on page 278
11. "WRITE Statement—NAMELIST with External Devices" on page 280
12. "WRITE Statement—NAMELIST with Internal Files" on page 282

End of IBM Extension

IBM Extension

WRITE Statement—Asynchronous

The asynchronous WRITE statement transmits data from an array in main storage to an external file.

Syntax

WRITE ([UNIT=]*un*, ID=*id*) *list*

UNIT=*un*
un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters 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.

list
is an asynchronous I/O list that may have any of four forms:

e
*e*1...*e*2
*e*1...
...*e*2

where:

e
is the name of an array.

***e*1 and *e*2**
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 sequential or direct access device. The array or array elements specified by *e* (or *e*1 and *e*2) 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 *e*1...*e*2 is specified, the transmitting area begins at array element *e*1 and includes every element up to and including *e*2. The subscript value of *e*1 must not exceed that of *e*2.

- If *e*1... is specified, the transmitting area begins at element *e*1 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 ...*e*2 is specified, the transmitting area begins at the first element of the array and includes every element up to and including *e*2.

- 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 the next 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.

Because the asynchronous WRITE statement can only refer to files with sequential access, REC may not be specified, even though the file may be resident on a direct-access device.

There is no FORMAT statement associated with the output data, and no conversion takes place.

Any number of program statements may be executed between an asynchronous WRITE and its corresponding WAIT, subject to the following rules:

- No such statement may in any way assign a new value to any array element in the transmitting field. This and the following rules apply also to indirect references to such array elements; that is, assigning a new value to a variable or array elements associated by COMMON or EQUIVALENCE statements, or argument association with an array element in the transmitting area.

- No executable statement may appear that redefines or undefines a variable or array element appearing in the subscript of *e*1 or *e*2.

- If a function reference appears in the subscript expression of *e*1 or *e*2, the function may not be referred to by any statements executed between the asynchronous WRITE and the corresponding WAIT. Also, no subroutines or function may be referred to that directly or indirectly refer to the subscript function, or to which the subscript function directly or indirectly refers.

- No function or subroutine may be executed that performs input or output on the file being manipulated.

Valid WRITE Statement:

```
WRITE (ID=10, UNIT=2*IN+2) . . . EXPECT(9)
```

End of IBM Extension

WRITE (Formatted, Direct Access)

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 opened for direct access. (See "OPEN Statement" on page 168.)

Syntax

WRITE ([UNIT=]*un*, [FMT=]*fmt*, REC=*rec* [, ERR=*stn*]

[, IOSTAT=*ios*]) [*list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

FMT=*fmt*

fmt is a required format identifier. It can, optionally, be preceded by FMT=.

If FMT= is not specified, the format identifier must appear second. If both UNIT= and FMT= are specified, all parameters, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- A statement number
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression

IBM Extension

- An array name

End of IBM Extension

The *statement number* must be the statement number of a FORMAT statement in the same program unit as the WRITE statement.

The *integer variable* must have been initialized by an ASSIGN statement with the number of a FORMAT statement. The FORMAT statement must be in the same program unit as the WRITE statement.

The *character constant* must constitute a valid format. The constant must be delimited by apostrophes, must begin with a left parenthesis, and must end with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. An apostrophe in a constant enclosed in apostrophes is represented by two consecutive apostrophes.

The *character variable* and *character array element* must contain character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. Blank characters may precede the left parenthesis and character data may follow the right parenthesis. The length of the format specification must not exceed the length of the character array element.

The *character array name* must contain character data whose leftmost characters constitute a valid format specification. The length of the format specification may exceed the length of the first element of the array; it is considered the concatenation of all the elements of the array in the order given by array element ordering.

IBM Extension

The *array name* may be of integer, real, double precision, logical, or complex type.

The data must be a valid format identifier as described under character array name above.

End of IBM Extension

The *character expression* may contain concatenations of character constants, character array elements, and character array names. Its value must be a valid format specification. The operands of the expression must have length specifications that contain only integer constants or names of integer constants.

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=*stn*
stn is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. Its value is positive if an error is detected, zero if no error is detected. VSAM return and reason codes are placed in *ios*.

list
is an I/O list and can contain variable names, array element names, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 92.

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 (un,fmt,REC=rec) list

WRITE (un,FMT=fmt,REC=rec) list

WRITE (FMT=fmt,REC=rec,UNIT=un) list

WRITE (REC=1, UNIT=11, FMT='(I9)')

WRITE (0,'(A8)', REC=3)
```

If this WRITE 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 OPEN is performed to a default file name. 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, or L, or is empty (that is, FORMAT( )), the record is filled with blank characters and the relative record number *rec* is increased by one.

IBM Extension

This is also true when the format specification starts with a Q or Z format code.

End of IBM Extension

Data and I/O List: The length of every VS FORTRAN record is specified in the RECL parameter 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.

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 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 Figure 39 on page 334.)

IBM Extension

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 opened for keyed access. (See "OPEN Statement" on page 168.)

Syntax

WRITE ([UNIT=]*un*, [FMT=]*fmt*, [, ERR=*stn*] [, IOSTAT=*ios*]

[, DUPKEY=*stn*]) *list*

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

FMT=*fmt*

fmt is a required format identifier. It can, optionally, be preceded by FMT=.

If FMT= is not specified, the format identifier must appear second. If both UNIT= and FMT= are specified, all parameters, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- A statement number
- 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 number* must be the statement number of a FORMAT statement in the same program unit as the WRITE statement.

The *integer variable* must have been initialized by an ASSIGN statement with the number of a FORMAT statement. The FORMAT statement must be in the same program unit as the WRITE statement.

The *character constant* must constitute a valid format. The constant must be delimited by apostrophes, must begin with a left parenthesis and end with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. An apostrophe in a constant enclosed in apostrophes is represented by two consecutive apostrophes.

The *character variable* and *character array element* must contain character data whose leftmost character positions constitute a valid format. A valid format begins with a left parenthesis and ends with a right parenthesis. Only the format codes described in the FORMAT statement can be used between the parentheses. Blank characters may precede the left parenthesis and character data may follow the right parenthesis. The length of the format specification must not exceed the length of the character array element.

The *character array name* must contain character data whose leftmost characters constitute a valid format specification. The length of the format specification may exceed the length of the first element of the array; it is considered the concatenation of all the elements of the array in the order given by array element ordering.

The *array name* may be of integer, real, double precision, logical, or complex type.

The data must be a valid format identifier as described under character array name above.

The *character expression* may contain concatenations of character constants, character array elements, and character array names. Its value must be a valid format specification. The operands of the expression must have length specifications that contain only integer constants or names of integer constants.

ERR=*stn*
: *stn* is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

IOSTAT=*ios*
: *ios* is an integer variable or an integer array element of length 4. Its value is positive if an error is detected, zero if no error is detected. VSAM return and reason codes are placed in *ios*.

DUPKEY=*stn*
: *stn* is the number of a statement to which control is passed when a duplicate-key condition occurs. See "Duplicate Key" below for an explanation of this condition.

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 92.

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

WRITE (Formatted, Keyed Access)

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 parameter 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 parameter 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 parameter 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 which 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.

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 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 Figure 39 on page 334.)

Duplicate Key: Control is transferred to the statement specified by DUPKEY when the a duplicate-key condition occurs; namely:

- The file is connected by an OPEN statement with an ACTION parameter 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
```

End of IBM Extension

WRITE (Formatted, Sequential Access)

WRITE Statement—Formatted with Sequential Access

This statement transfers data from internal storage onto an external I/O 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 is connected with sequential access to a unit. (See "OPEN Statement" on page 168.)

Syntax

WRITE ([UNIT=]*un*, [FMT=]*fmt* [, ERR=*stn*] [, IOSTAT=*ios*])

[*list*]

PRINT *fmt* [, *list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is either:

- An integer expression of length 4 whose value must be zero or positive
- An asterisk (*) representing an installation-dependent unit

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified all the parameters can appear in any order.

In the form of a PRINT statement where *un* cannot be specified, *un* is installation dependent.

FMT=*fmt*

fmt is a required format identifier. It can, optionally, be preceded by FMT=.

If FMT is not specified, the format identifier must appear second. If both UNIT= and FMT= are specified, all parameters, except *list*, can appear in any order.

The format identifier (*fmt*) can be:

- A statement number
- An integer variable
- A character constant
- A character variable
- A character array element
- A character array name
- A character expression

IBM Extension

An array name

End of IBM Extension

See "WRITE Statement—Formatted with Direct Access" on page 250 for explanations of these format identifiers.

ERR=*stn*
stn is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

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

list
is an I/O list. It can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. In the PRINT statement, if the list is not present, the comma must be omitted. See "Implied DO in an Input/Output Statement" on page 92.

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

Valid WRITE and PRINT Statements:

```
WRITE (un,fmt) list

WRITE (un, FMT=fmt) list

WRITE (*,fmt) list

WRITE (UNIT=un, FMT=fmt) list     FMT=fmt can appear first.

WRITE(IOSTAT=IOS,ERR=99999,FMT=*,UNIT=2*IN+3)

WRITE(IN+8,NAMEOT,IOSTAT=IACT(1),ERR=99999)

PRINT *, list

PRINT fmt, list

PRINT fmt
```

Invalid WRITE and PRINT Statements:

WRITE (fmt,un)	un must appear first before fmt.
WRITE (FMT=fmt,un) list	un must appear first because UNIT= is not specified.
WRITE (fmt,UNIT=un) list	FMT= must be used because UNIT= is specified.
PRINT FMT=fmt, list	FMT= must not be used with PRINT.

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: 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, or L, or is empty (that is, FORMAT( )), a blank record is written out.

IBM Extension

This is also true when the format specification starts with a Q or Z format code.

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.

End of IBM Extension

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 108.)

VS FORTRAN Programming Guide describes how to associate FORTRAN records (that is, logical records) and physical records on an external I/O device.

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 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 Figure 39 on page 334.)

WRITE (Unformatted, Direct Access)

WRITE Statement—Unformatted with Direct Access

This statement transfers data without conversion from internal storage onto an external I/O device. 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=*stn*] [, IOSTAT=*ios*]

[, NUM=*n*]) *list*

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

REC=*rec*

rec is an integer expression. It represents the relative position of a record within the file associated with *un*. Its value after conversion to integer, if necessary, must be greater than zero. The internal record number of the first record is 1. The INQUIRE statement can be used to determine the record number.

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=*stn*

stn is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

IOSTAT=*ios*

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

IBM Extension

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 parameter 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.

End of IBM Extension

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 92.

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 (un,REC=rec) list

WRITE (REC=rec,UNIT=un) list

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. If the unit is not connected to a file, it is assumed to have been preconnected through job control language, and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

Data Transmission: A WRITE 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 specified in the RECL parameter 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 parameter 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 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 Figure 39 on page 334.)

IBM Extension

WRITE Statement—Unformatted with Keyed Access

This statement transfers data without conversion from internal storage onto an external I/O device. 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=*stn*] [, IOSTAT=*ios*]

  [, DUPKEY=*stn*] [, NUM=*n*]) *list*

UNIT=*un*
: *un* is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

  It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*stn*
: *stn* is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

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

DUPKEY=*stn*
: *stn* is the number of a statement to which control is passed when a duplicate-key condition occurs. See "Duplicate Key" below for an explanation of this condition.

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 parameter 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 92.

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 parameter 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 parameter 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 parameter 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 which 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.

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 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 Figure 39 on page 334.)

Duplicate Key: Control is transferred to the statement specified by DUPKEY when the a duplicate-key condition occurs; namely:

- The file is connected by an OPEN statement with an ACTION parameter of 'READWRITE', or when ACTION='WRITE', and
- An attempt was made to write a record with a key whose values must be unique, and
- The key value would have duplicated one that already exists for the same key in another record.

If IOSTAT=*ios* is specified, a positive integer value is assigned to *ios* when the duplicate-key condition is detected. If ERR is specified but DUPKEY is not, control passes to the statement specified by ERR when the duplicate-key condition is detected. If neither DUPKEY nor ERR was given, an error is detected.

Examples:

```
WRITE (UNIT=10) AA, BB, CC
WRITE (10,DUPKEY=77) AA, BB, CC
WRITE (10,NUM=LENG) AA, BB, CC
```

End of IBM Extension

WRITE Statement—Unformatted with Sequential Access

This statement transfers data without conversion from internal storage onto an external I/O device. The data must be sent to an external file that is connected with sequential access to a unit (see "OPEN Statement" on page 168).

Syntax

WRITE ([UNIT=]*un* [, ERR=*stn*] [, IOSTAT=*ios*]

[, NUM=*n*]) [*list*]

UNIT=*un*
un is the reference to the number of an I/O unit. *un* is an integer expression of length 4 whose value must be zero or positive.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

ERR=*stn*
stn is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

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

IBM Extension

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 parameter 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.

End of IBM Extension

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 92.

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 (un) list

WRITE (UNIT=un) list

WRITE(5) EXPECT(4)
```

Invalid WRITE Statement:

```
WRITE un,list              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.

IBM Extension

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.

End of IBM Extension

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 Programming Guide describes how to associate FORTRAN records (that is, logical records) and physical records on an external I/O device.

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 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 Figure 39 on page 334.)

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=*stn*] [, IOSTAT=*ios*])

[*list*]

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It can be the name of a character variable, character array, character array element, or character substring.

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, FMT= must be used, and all the parameters can appear in any order.

FMT=*fmt*

is the format specification. It may, optionally, be preceded by FMT=.

If FMT= is not specified, the format specification must appear second. If both UNIT= and FMT= are specified, all parameters, except *list*, may appear in any order.

The format specification can be:

- A statement number
- An integer variable
- A character constant
- A character variable
- A character array name
- A character array element
- A character expression

IBM Extension

- An array name

End of IBM Extension

See "WRITE Statement—Formatted with Direct Access" on page 250 for explanations of these format specifications.

ERR=*stn*
: *stn* is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

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

list
: is an I/O list and can contain variable names, array elements, character substring names, array names (except the names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 92.

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*
- Associated with any part of *un* through EQUIVALENCE, COMMON, or argument passing

Valid WRITE Statements:

```
CHARACTER *5 CHAR
DIMENSION IACT (10)

WRITE (un,fmt) list

WRITE (un,FMT=fmt) list

WRITE (FMT=fmt,UNIT=un) list

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,un) list          un must appear first before fmt.

WRITE (FMT=fmt,un) list      un must appear first because
                             UNIT= is not specified.

WRITE (fmt,UNIT=un) list     FMT= must be used because UNIT=
                             is specified.
```

Data Transmission: A WRITE statement starts data transmission at the beginning of the area specified by *un*. The format codes in the format specification *fmt* are taken one by one and associated with every item of the list in the order they are specified. Data is taken from the item of the list, converted according to the format code, and the number of character data specified by the format code is moved into the storage area *un*. Data transmission stops when data has been moved from every item of the list.

If *un* is a character variable, a character array element, or a character substring name, it is treated as one record only in relation to the format specification.

If *un* is a character array name, each array element is treated as one record in relation to the format specification.

If the list is not specified and the format specification starts with an I, E, F, D, G, or L, or is empty (that is, FORMAT( )), the record is filled with blank characters and the relative record number *rec* is increased by one.

IBM Extension

This is also true when the format specification starts with a Q or Z format code.

End of IBM Extension

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.

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 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 Figure 39 on page 334.)

Valid Internal File Examples:

The following example illustrates how to use an internal READ of a character variable to initialize an integer array.

```
CHARACTER*24 CHAR
INTEGER IARRY(3,4)
```

Initialize the character variable CHAR.

```
    READ(5,'(A24)') CHAR
```

Assume that the data read into CHAR is:

010203040506070809101112

Now, the program will use CHAR as an internal file and read it to initialize IARRY.

```
    READ (CHAR,10) ((IARRY(I,J), I = 1,3),J=1,4)
 10 FORMAT(12I2)
    PRINT *,IARRY
    STOP
    END
```

The following example illustrates how to convert an integer number to its character representation. This example also illustrates a technique for changing a FORMAT statement dynamically; that is, the example initializes the specification of the field width for the A edit descriptor.

```
     CHARACTER*8 FMT
     DATA FMT /'(1X,AYY)'/
     I = 4
     WRITE (FMT(6:7), 10) I
10   FORMAT (I2)
     ...
     PRINT 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 onto an external I/O device. 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=*stn*] [, IOSTAT=*ios*])

[*list*]

PRINT * [, *list*]

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is either:

- An integer expression of length 4 whose value must be zero or positive
- An asterisk (*) representing an installation-dependent unit

It is required and can optionally be preceded by UNIT=.

If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, FMT= must be used, and all the parameters, except *list*, can appear in any order.

In the form of a PRINT statement where *un* cannot be specified, *un* is installation dependent.

FMT=*

An asterisk (*) specifies that a list-directed WRITE has to be executed. It can, optionally, be preceded by FMT= if *un* is specified.

If FMT= is not specified, the format identifier must appear second. If both UNIT= and FMT= are specified, all parameters, except *list*, may appear in any order.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

IOSTAT=*ios*

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

list

is an I/O list and can contain variable names, array elements, character substring names, array names (except names of assumed-size arrays), implied DO lists, and expressions. See "Implied DO in an Input/Output Statement" on page 92.

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 (un,*) list

WRITE (un,FMT=*) list

WRITE (FMT=*,UNIT=un) list

WRITE (5,*)

WRITE (FMT=*,UNIT=*) FIFTY5,ISEG

WRITE (IOSTAT=IOS, ERR=99999, FMT=*, UNIT=2*IN+3)
       ''''//EXPECT(1)//''''

PRINT *, list
```

Invalid WRITE Statements:

`WRITE (*,un) list`	`un must appear first because UNIT= is not specified.`
`WRITE (FMT=*,un) list`	`un must appear first because UNIT= is not specified.`
`WRITE(*,UNIT=un) list`	`FMT= must be used because UNIT= is specified.`
`PRINT FMT=*, list`	`FMT= must not be used with the second form of syntax.`

If the unit specified by *un* is encountered, it must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

Data Transmission: A WRITE or PRINT statement with list-directed I/O accessing an external file starts data transmission at the beginning of a record. The data is taken from each item in the list in the order they are specified and transmitted onto the record of the external file. Data transmission stops when data has been transferred from every item in the list.

After execution of a sequential WRITE or PRINT statement, no record exists in the file following the last record transferred by that statement.

The WRITE or PRINT statement can write over an end of file and extend the external file. An ENDFILE, CLOSE, or REWIND statement will reinstate the end of file.

An external file with sequential access written with list-directed I/O is suitable *only* for printing, because a blank character is always inserted at the beginning of each record as a carrier 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 VS FORTRAN record (also called a logical record). A single WRITE or PRINT statement creates only one VS FORTRAN record.

For information on how to calculate the size of a record needed to hold all the converted list items, see Figure 20. It shows the width of the written field for any item's data type and length. The size of the record will be the sum of the field widths plus a byte to separate each field.

Data Type	Length	Field Width
Real	16	42 bytes
Real	8	25 bytes
Real	4	16 bytes
Logical	1 or 4	1 byte
Integer	2	6 bytes
Integer	4	11 bytes
Complex	32	84 bytes
Complex	16	51 bytes
Complex	8	25 bytes
Character	*	132 bytes (See Note)

Figure 20. Field Widths Needed for Data Types of Various Lengths

Note to Figure 20: The number of bytes printed out is determined by the size of the character type item. The number of characters per record is determined by the type of data set being written to. The number of bytes per record is determined by the logical record length. For output that is sent to a terminal, a carriage control character is deleted at the beginning of each record. This is also true for a file defined with a carriage control character. Character data can be split between records. Numeric data cannot be split between records.

VS FORTRAN Programming Guide describes how to associate FORTRAN records (that is, logical records) and physical records on an external I/O device. In particular, a logical record may span many physical records. A character constant or a complex constant can be split over the next physical record if there is not enough space on the current physical record to contain it all.

Character constants produced:

- Are not delimited by apostrophes
- Are not preceded or followed by any separators (including blanks)
- Have each internal apostrophe represented externally by one apostrophe
- Have a blank character inserted by the processor for carrier control at the beginning of any record that begins with the continuation of a character constant from the preceding record

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 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 Figure 39 on page 334.)

IBM Extension

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=*stn*] [, IOSTAT=*ios*]) [*list*]

UNIT=*un*

un is the reference to an area of internal storage called an internal file. It can be the name of:

- A character variable
- A character array
- A character array element
- A character substring

It is required and can, optionally, be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

FMT=*

* specifies that a list-directed WRITE is to be executed. It can, optionally, be preceded by FMT=.

If FMT= is not specified, * must appear second. If both UNIT= and FMT= are specified, all the parameters can appear in any order.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

IOSTAT=*ios*

ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected, negative if an end of file is encountered, and zero if no error condition is detected.

list

is an I/O list and can contain variable names, array element names, character substring names, array names (except names of assumed-size arrays), and implied DO lists. See "Implied DO in an Input/Output Statement" on page 92.

Valid WRITE Statements:

```
WRITE (un,*) list
WRITE (un,FMT=*) list
WRITE (FMT=*,UNIT=un) list
WRITE (IOSTAT=IACT(1), UNIT=CHARVR, FMT=*) ACTUAL(1)
```

Data Transmission: An internal WRITE statement starts data transmission at the beginning of the storage area specified by *un*. Each item of the list is transferred to the internal file in the order it is specified. The conversion to be performed depends on the type and length of the name of the item in the list. Data transmission stops when every item of the list has been moved to the internal file or when the end of the internal file is reached.

Data and I/O List: If *un* is a character variable, a character array element name, or a character substring name, it is treated as one record. If *un* is a character array name, each array element is treated as one record. If a record is not large enough to hold all the converted items, a new record is started for any noncharacter item that will exceed the record length. For character items, as much as can be put in the record is written there, and the remainder is written at the beginning of the next record.

The length of a record is the length of the character variable, character substring name, character array element specified by *un* when the WRITE statement is executed.

For information on how to calculate the size of a record needed to hold all the converted list items, see Figure 20 on page 276. It shows the width of the written field for any item's data type and length. The size of the record will be the sum of the field widths plus a byte to separate each field.

```
    CHARACTER* 120 CHARVR

    WRITE (UNIT=CHARVR, FMT=*) A1, A2, A3

100 FORMAT (A120)

    WRITE (UNIT=6, FMT=100) CHARVR
```

Statement 1 defines a character variable, CHARVR, of fixed-length 120. Statement 2 writes the internal file represented by CHARVR by converting the values in A1, A2, and A3. Statement 3 writes the 120 characters of output onto an external file.

End of IBM Extension

WRITE (NAMELIST, External)

IBM Extension

WRITE Statement—NAMELIST with External Devices

This statement transfers data from internal storage onto an external I/O device. The type of the items specified in the NAMELIST statement determines the conversions to be performed.

Syntax

WRITE ([UNIT=]*un*, [FMT=]*name* [, ERR=*stn*] [, IOSTAT=*ios*])

PRINT *name*

UNIT=*un*

un is the reference to the number of an I/O unit. *un* is one of the following:

- An integer expression of length 4 whose value must be zero or positive
- An asterisk (*) representing an installation-dependent unit

un is required in the first form of the WRITE statement and can optionally be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. The other parameters may appear in any order. If UNIT= is specified, all the parameters can appear in any order.

In the form of the WRITE where *un* is not specified, *un* is installation dependent.

FMT=*name*

name is a NAMELIST name. See "NAMELIST Statement" on page 166.

If FMT= is not specified, the NAMELIST name must appear second. If both UNIT= and FMT= are specified, all the parameters can appear in any order.

ERR=*stn*

stn is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

IOSTAT=*ios*

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

Valid WRITE Statements:

```
WRITE (un, name)

WRITE (IN+8, NAMEOUT, IOSTAT=IACT(1), ERR=99999)
```

Invalid WRITE Statements:

```
WRITE (name,un)          un must appear before name.

WRITE (un,name) list     list must not be specified.
```

If the unit specified by *un* is encountered, it must exist and must be connected to a file for sequential access. If the unit is not connected to a file, it is assumed to have been preconnected through job control language and an implicit OPEN is performed to a default file name. If the file is not preconnected, an error is detected.

A BACKSPACE or REWIND statement should not be used for a file that is written using NAMELIST. If it is, the results are unpredictable (see "BACKSPACE Statement" on page 67).

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.

For information on how to calculate the size of the record on the external file, see Figure 20 on page 276. It shows the width of the written field for any item's data type and length. The size of the record will be the sum of the field widths plus:

- The number of bytes needed for each item's name and an equal sign (these are prefixed to each field), and
- A byte to separate each field

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 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 Figure 39 on page 334.)

End of IBM Extension

IBM Extension

WRITE Statement—NAMELIST with Internal Files

This statement transfers data from one or more areas of internal storage to another area of internal storage. The receiving area is called an internal file. This statement can be used to convert numeric data to character data. The type of the items specified in an associated NAMELIST list determines the conversions to be performed.

Syntax

WRITE ([UNIT=]*un*, [FMT=]*name* [, ERR=*stn*] [, IOSTAT=*ios*])

UNIT=*un*
un is the reference to an area of internal storage called an internal file. It must be the name of a character array with at least three elements.

It is required and can optionally be preceded by UNIT=. If UNIT= is not specified, *un* must appear first in the statement. If UNIT= is specified, FMT= must be used and all the parameters can appear in any order.

FMT=*name*
name is a NAMELIST name. See "NAMELIST Statement" on page 166.

If FMT= is not specified, the NAMELIST name must appear second. If both UNIT= and FMT= are specified, all the parameters can appear in any order.

ERR=*stn*
stn is the number of an executable statement in the same program unit as the WRITE statement. If an error is detected, control is transferred to *stn*.

IOSTAT=*ios*
ios is an integer variable or an integer array element of length 4. *ios* is set positive if an error is detected; it is set to zero if no error condition is detected.

Valid WRITE Statements:

```
WRITE (un, name)

WRITE (IN+8, NAMOUT, IOSTAT=IACT(1), ERR=99999)
```

Data Transmission: A WRITE statement with NAMELIST starts data transmission from the beginning of the internal file. The data is taken from each item in the list associated with the NAMELIST name, in the order in which the items are specified, and transmitted to the internal file. Data transmission stops when data has been transferred from every item in the list.

Data and NAMELIST: The NAMELIST name must appear in the internal file.

The number of characters specified by the items in the NAMELIST name and the necessary data separators and identifiers are placed in the internal file.

For information on how to calculate the size of the internal file, see Figure 20 on page 276. It shows the width of the written field for any item's data type and length. The size of the internal file will be the sum of the field widths plus:

- The number of bytes needed for each item's name and an equal sign (these are prefixed to each field), and
- A byte to separate each field.

Example:

```
NAMELIST  /NL1/I,J,C
CHARACTER*40 CHAR(3)
CHARACTER*5  C
INTEGER*2    I,J
I=12046
J=12047
C='BACON'
WRITE(CHAR,NL1)
```

After execution of the WRITE statement:

```
                 Position 2
                 v
CHAR(1) contains &NL1
CHAR(2) contains I= 12046,J= 12047,C='BACON'
CHAR(3) contains &END
```

End of IBM Extension

Chapter 6. VS FORTRAN Intrinsic Functions

Intrinsic functions are procedures supplied in VS FORTRAN for standard mathematical computations and bit manipulations. A procedure is invoked by including its name in an arithmetic or character expression accompanied by one or more arguments. The compiler recognizes the procedure by its name, checks the syntax of the arguments, and generates code that performs the desired function.

The general format for referring to an intrinsic function is

```
name (arg1[,arg2...,argn])
```

where *name* is the function name and *arg1*, *arg2*, and *argn* are the actual arguments.

For example, the source statement

```
SINRAD=SIN(RADIAN)
```

causes the sine function to be invoked. The value of the argument RADIAN is given to the sine function, which computes the sine of that value. The result is stored in the variable SINRAD.

Nearly all the mathematical functions have both generic and specific names. Use of the generic name simplifies the referencing of the functions because the same name may be used for the entire range of argument types permitted. The appropriate specific entry name is chosen (by the compiler) when the generic name is used, based on the type of argument(s) presented.

Figure 28 on page 301 lists all the generic function names and gives the valid range of argument types and function values.

The intrinsic functions provided by VS FORTRAN are described in detail in the following figures, grouped by function:

Function	**Figure**
Logarithmic and Exponential	Figure 21 on page 288
Trigonometric	Figure 22 on page 289
Hyperbolic	Figure 23 on page 291
Miscellaneous Mathematical	Figure 24 on page 292
Conversion and Maximum/Minimum	Figure 25 on page 296

Function	Figure
Character Manipulation	Figure 26 on page 299
Bit Manipulation	Figure 27 on page 300

All the specific function names listed in Figure 21 through Figure 24, and in Figure 27, can be passed as actual arguments. None of the function names listed in Figure 25 or Figure 26 (except for LEN and INDEX) can be passed as actual arguments. (An INTRINSIC statement for a specific function name must appear in any program unit that passes the name as an actual argument.)

References to the functions are either resolved from the library or inserted in the object module. That is, the code generated by VS FORTRAN for the reference contains either instructions to link to the function in the library (out-of-line) or instructions to perform the function directly (inline). Notes with the figures state whether the functions are performed inline or out-of-line.

For a small subset of the mathematical functions, alternative procedures are available that under certain conditions provide greater accuracy and faster computation. These functions are identified in footnotes in the figures. For more information, see Chapter 8, "Mathematical, Character, and Bit Subprograms" on page 307.

The following information is provided for each entry name in Figure 21 on page 288 through Figure 27 on page 300:

General Function: This column states the nature of the computation performed by the function.

Generic Name This column gives the generic name of the function (if any).

Entry Name: This column gives the specific entry names of the function. A function may have more than one entry name; the particular entry name used depends on the computation to be performed. For example, the sine and cosine function has two entry names: SIN and COS. If the sine is to be computed, entry name SIN is used; if the cosine is to be computed, entry name COS is used.

Definition: This column gives a mathematical equation that represents the computation. An alternate equation is given in those cases in which there is another way of representing the computation in mathematical notation. For example, the square root can be represented either as:

$y = \sqrt{x}$ or $y = x^{1/2}$

Argument Number: This column states how many arguments the programmer must supply.

Argument Type: This column describes the type and length of each of the argument(s). INTEGER, REAL, COMPLEX, LOGICAL, and Character represent the type; the notations *1, *4, *8, *16, *32, and *n represent the size of the argument in number of storage locations. (The notation *n describes character data.)

Argument Range: This column gives the valid range for arguments. If an argument is not within the range, an error message is issued (see Error Code column).

Function Value Type and Range: This column describes the type and range of the function value returned by the subprogram. Type notation used is the same as that for the argument type. The range symbol is

$$\gamma = 16^{63} (1 - 16^{-6})$$

for regular precision routines;

$$\gamma = 16^{63} (1 - 16^{-14})$$

for double-precision; and

$$\gamma = 16^{63} (1 - 16^{-28})$$

for extended precision.

Error Code: This column gives the number of the message issued when an error occurs. Appendix I, "Library Procedures and Messages" on page 463 contains descriptions of the error messages.

Throughout these figures, the following approximate values are represented by

$(2^{18} \cdot \pi)$ and $(2^{50} \cdot \pi)$:

$$(2^{18} \cdot \pi) = .8235496645826428D + 06$$
$$(2^{50} \cdot \pi) = .3537118876014220D + 16$$

General Function[6]	Entry Name	Definition	Argument(s) No.	Argument(s) Type[1]	Argument(s) Range	Function Value Type[1] and Range[4]	Error Code
Common and natural logarithm	ALOG	$y=\log_e x$ or $y=\ln x$	1	REAL *4	$x > 0$	REAL *4 $y \geqq -180.218$ $y \leqq 174.673$	253
	ALOG10	$y=\log_{10} x$	1	REAL *4	$x > 0$	REAL *4 $y \geqq -78.268$ $y \leqq 75.859$	253
	DLOG	$y=\log_e x$ or $y=\ln x$	1	REAL *8	$x > 0$	REAL *8 $y \geqq -180.218$ $y \leqq 174.673$	263
	DLOG10	$y=\log_{10} x$	1	REAL *8	$x > 0$	REAL *8 $y \geqq -78.268$ $y \leqq 75.859$	263
	CLOG	$y=$ PV $\log_e (z)$ See Note 2	1	COMPLEX *8	$z \neq 0 + 0i$	COMPLEX *8 $y_1 \geqq -180.218$ $y_1 \leqq 175.021$ $-\pi \leqq y_2 \leqq \pi$	273
	CDLOG	$y=$ PV $\log_e (z)$ See Note 2	1	COMPLEX *16	$z \neq 0 + 0i$	COMPLEX *16 $y_1 \geqq -180.218$ $y_1 \leqq 175.021$ $-\pi \leqq y_2 \leqq \pi$	283
	QLOG	$y=\log_e x$ or $y=\ln x$	1	REAL *16	$x>0$	REAL *16 $y \geqq -180.218$ $y \leqq 174.673$	293
	QLOG10	$y=\log_{10} x$	1	REAL *16	$x>0$	REAL *16 $y \geqq -78.268$ $y \leqq 175.859$	293
	CQLOG	$y=$ PV $\log_e (z)$ See Note 2	1	COMPLEX *32	$z \neq 0 + 0i$	COMPLEX *32 $y_1 \geqq -180.218$ $y_1 \leqq 175.021$ $-\pi \leqq y_2 \leqq \pi$	278
Exponential	EXP	$y=e^x$ See Note 5	1	REAL *4	$x \leqq 174.673$	REAL *4 $0 \leqq y \leqq \gamma$	252
	DEXP	$y=e^x$ See Note 5	1	REAL *8	$x \leqq 174.673$	REAL *8 $0 \leqq y \leqq \gamma$	262
	CEXP	$y=e^z$ See Note 3	1	COMPLEX *8	$x_1 \leqq 174.673$ $\lvert x_2 \rvert < (2^{18} \cdot \pi)$	COMPLEX *8 $-\gamma \leqq y_1, y_2 \leqq \gamma$	271, 272
	CDEXP	$y=e^z$ See Note 3	1	COMPLEX *16	$x_1 \leqq 174.673$ $\lvert x_2 \rvert < (2^{50} \cdot \pi)$	COMPLEX *16 $-\gamma \leqq y_1, y_2 \leqq \gamma$	281, 282
	QEXP	$y=e^x$	1	REAL *16	$x \geqq -180.218$ $x \leqq 174.673$	REAL *16 $0 \leqq y \leqq \gamma$	292
	CQEXP	$y=e^z$ See Note 3	1	COMPLEX *32	$x_1 \leqq 174.673$ $x_2 \leqq 2^{100}$	COMPLEX *32 $-\gamma \leqq y_1, y_2 \leqq \gamma$	276, 277

NOTES:

[1] REAL *4, REAL *8, and REAL *16 arguments correspond to REAL, DOUBLE PRECISION, and EXTENDED PRECISION arguments, respectively, in VS FORTRAN.

[2] PV = principal value. The answer given $(y_1 + y_2 i)$ is that one whose imaginary part (y_2) lies between $-\pi$ and $+\pi$. More specifically: $-\pi < y_2 \leqslant \pi$, unless $x_1 < 0$ and $x_2 = -0$, in which case, $y_2 = -\pi$.

[3] z is a complex number of the form $x_1 + x_2 i$.

[4] $\gamma = 16^{63}(1 - 16^{-6})$ for regular precision routines, $16^{63}(1 - 16^{-14})$ for double precision routines, and $16^{63}(1 - 16^{-28})$ for extended precision.

[5] Available also in the alternative mathematical library.

[6] All functions are generated as out-of-line library calls.

Figure 21. Logarithmic and Exponential Functions

General Function[6]	Entry Name	Definition	Argument(s)			Function Value Type[1] and Range[5]	Error Code
			No.	Type[1]	Range		
Arcsine and arccosine	ASIN	y=arcsin (x)	1	REAL *4	$\|x\| \leqq 1$	REAL *4 (in radians) $-\frac{\pi}{2} \leqq y \leqq \frac{\pi}{2}$	257
	ACOS	y=arccos (x)	1	REAL *4	$\|x\| \leqq 1$	REAL *4 (in radians) $0 \leqq y \leqq \pi$	257
	DASIN	y=arcsin (x)	1	REAL *8	$\|x\| \leqq 1$	REAL *8 (in radians) $-\frac{\pi}{2} \leqq y \leqq \frac{\pi}{2}$	267
	DACOS	y=arccos (x)	1	REAL *8	$\|x\| \leqq 1$	REAL *8 (in radians) $0 \leqq y \leqq \pi$	267
	QARSIN	y=arcsin(x)	1	REAL*16	$\|x\| \leqq 1$	REAL*16 $-\frac{\pi}{2} \leqq y \leqq \frac{\pi}{2}$	297
	QARCOS	y=arcos(x)	1	REAL*16	$\|x\| \leqq 1$	REAL*16 $0 \leqq y \leqq \pi$	297
Arctangent	ATAN	y=arctan (x)	1	REAL *4	any REAL argument	REAL *4 (in radians) $-\frac{\pi}{2} \leqq y \leqq \frac{\pi}{2}$	None
	ATAN2	$y=\arctan\left(\frac{x_1}{x_2}\right)$	2	REAL *4	any REAL arguments (except 0, 0)	REAL *4 (in radians) $-\pi < y \leqq \pi$	255
	DATAN	y=arctan (x)	1	REAL *8	any REAL argument	REAL *8 (in radians) $-\frac{\pi}{2} \leqq y \leqq \frac{\pi}{2}$	None
	DATAN2	$y=\arctan\left(\frac{x_1}{x_2}\right)$	2	REAL *8	any REAL arguments (except 0, 0)	REAL *8 (in radians) $-\pi < y \leqq \pi$	265
	QATAN	y=arctan(x)	1	REAL*16	any REAL argument	REAL*16 (in radians) $-\frac{\pi}{2} \leqq y \leqq \frac{\pi}{2}$	None
	QATAN2	$y=\arctan\left(\frac{X_1}{X_2}\right)$	2	REAL*16	any REAL arguments (except 0,0)	REAL*16 (in radians) $-\pi < y \leqq \pi$	295
NOTES: (See end of table.)							

Figure 22 (Part 1 of 2). Trigonometric Functions

General Function[6]	Entry Name	Definition	Argument(s)			Function Value Type[1] and Range[5]	Error Code
			No.	Type[1]	Range		
Sine and cosine	SIN	y=sin (x)	1	REAL *4 (in radians)	$\|x\| < (2^{18} \cdot \pi)$	REAL *4 $-1 \leq y \leq 1$	254
	COS	y=cos (x)	1	REAL *4 (in radians)	$\|x\| < (2^{18} \cdot \pi)$	REAL *4 $-1 \leq y \leq 1$	254
	DSIN	y=sin (x)	1	REAL *8 (in radians)	$\|x\| < (2^{50} \cdot \pi)$	REAL *8 $-1 \leq y \leq 1$	264
	DCOS	y=cos (x)	1	REAL *8 (in radians)	$\|x\| < (2^{50} \cdot \pi)$	REAL *8 $-1 \leq y \leq 1$	264
	CSIN	y=sin (z) See Note 2	1	COMPLEX *8 (in radians)	$\|x_1\| < (2^{18} \cdot \pi)$ $\|x_2\| \leq 174.673$	COMPLEX *8 $-\gamma \leq y_1, y_2 \leq \gamma$	274, 275
	CCOS	y=cos (z) See Note 2	1	COMPLEX *8 (in radians)	$\|x_1\| < (2^{18} \cdot \pi)$ $\|x_2\| \leq 174.673$	COMPLEX *8 $-\gamma \leq y_1, y_2 \leq \gamma$	274, 275
	CDSIN	y=sin (z) See Note 2	1	COMPLEX *16 (in radians)	$\|x_1\| < (2^{50} \cdot \pi)$ $\|x_2\| \leq 174.673$	COMPLEX *16 $-\gamma \leq y_1, y_2 \leq \gamma$	284, 285
	CDCOS	y=cos (z) See Note 2	1	COMPLEX *16 (in radians)	$\|x_1\| < (2^{50} \cdot \pi)$ $\|x_2\| \leq 174.673$	COMPLEX *16 $-\gamma \leq y_1, y_2 \leq \gamma$	284, 285
	QSIN	y=sin(x)	1	REAL*16 (in radians)	$\|x\| < 2^{100}$	REAL*16 $-1 \leq y \leq 1$	294
	QCOS	y=cos(x)	1	REAL*16 (in radians)	$\|x\| < 2^{100}$	REAL*16 $-1 \leq y \leq 1$	294
	CQSIN	y=sin(z) See Note 2	1	COMPLEX*32 (in radians)	$\|x_1\| < 2^{100}$ $\|x_2\| \leq 174.673$	COMPLEX*32 $-\gamma \leq y_1, y_2 \leq \gamma$	279, 280
	CQCOS	y=cos(z) See Note 2	1	COMPLEX*32 (in radians)	$\|x_1\| < 2^{100}$ $\|x_2\| \leq 174.673$	COMPLEX*32 $-\gamma \leq y_1, y_2 \leq \gamma$	279, 280
Tangent and cotangent	TAN	y=tan (x)	1	REAL *4 (in radians)	$\|x\| < (2^{18} \cdot \pi)$ See Note 4	REAL *4 $-\gamma \leq y \leq \gamma$	258, 259
	COTAN	y=cotan (x)	1	REAL *4 (in radians)	$\|x\| < (2^{18} \cdot \pi)$ See Note 4	REAL *4 $-\gamma \leq y \leq \gamma$	258, 259
	DTAN	y=tan (x)	1	REAL *8 (in radians)	$\|x\| < (2^{50} \cdot \pi)$ See Note 4	REAL *8 $-\gamma \leq y \leq \gamma$	268, 269
	DCOTAN	y=cotan (x)	1	REAL *8 (in radians)	$\|x\| < (2^{50} \cdot \pi)$ See Note 4	REAL *8 $-\gamma \leq y \leq \gamma$	268, 269
	QTAN	y=tan(x)	1	REAL*16 (in radians)	$\|x\| < 2^{100}$ See Note 3	REAL*16 $-\gamma \leq y \leq \gamma$	298, 299
	QCOTAN	y=cotan(x)	1	REAL*16 (in radians)	$\|x\| < 2^{100}$ $\|x\| \geq 16^{-63}$ See Note 3	REAL*16 $-\gamma \leq y \leq \gamma$	298, 299

NOTES:

[1] REAL *4, REAL *8, and REAL *16 correspond to REAL, DOUBLE PRECISION, and EXTENDED PRECISION arguments, respectively, in VS FORTRAN.

[2] z is a complex number of the form $x_1 + x_2 i$.

[3] x may not be such that one can find a singularity within 8 units of the last digit value of the floating-point representation of x. Singularities are $\pm$ $(2n + 1)\frac{\pi}{2}$, n = 0, 1, 2, ... for tangent, and $\pm$ n π, n = 0, 1, 2, ... for cotangent.

[4] The argument for the cotangent functions may not approach a multiple of π; the argument for the tangent functions may not approach an odd multiple of $\pi/2$.

[5] $\gamma = 16^{63} (1 - 16^{-6})$ for regular precision routines, $16^{63} (1 - 16^{-14})$ for double-precision routines and $16^{63} (1 - 16^{-28})$ for extended precision.

[6] All functions are generated as out-of-line library calls.

Figure 22 (Part 2 of 2). Trigonometric Functions

General Function[3]	Entry Name	Definition	Argument(s)			Function Value Type[1] and Range[2]	Error Code
			No.	Type[1]	Range		
Hyperbolic sine and cosine	SINH	$y = \frac{e^x - e^{-x}}{2}$	1	REAL *4	$\|x\| < 175.366$	REAL *4 $-\gamma \leqq y \leqq \gamma$	256
	COSH	$y = \frac{e^x + e^{-x}}{2}$	1	REAL *4	$\|x\| < 175.366$	REAL *4 $1 \leqslant y \leqslant \gamma$	256
	DSINH	$y = \frac{e^x - e^{-x}}{2}$	1	REAL *8	$\|x\| < 175.366$	REAL *8 $-\gamma \leqslant y \leqslant \gamma$	266
	DCOSH	$y = \frac{e^x + e^{-x}}{2}$	1	REAL *8	$\|x\| < 175.366$	REAL *8 $1 \leqslant y \leqslant \gamma$	266
	QSINH	$y = \frac{e^x - e^{-x}}{2}$	1	REAL *16	$\|x\| \leqslant 175.366$	REAL *16 $-\gamma \leqslant y \leqslant \gamma$	296
	QCOSH	$y = \frac{e^x + e^{-x}}{2}$	1	REAL *16	$\|x\| \leqq 175.366$	REAL *16 $1 \leqq y \leqq \gamma$	296
Hyperbolic tangent	TANH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1	REAL *4	any REAL argument	REAL *4 $-1 \leqq y \leqq 1$	None
	DTANH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1	REAL *8	any REAL argument	REAL *8 $-1 \leqq y \leqq 1$	None
	QTANH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1	REAL *16	any REAL argument	REAL *16 $-1 \leqq y \leqq 1$	None

NOTES:

[1] REAL *4, REAL *8, and REAL *16 arguments correspond to REAL, DOUBLE PRECISION, and EXTENDED PRECISION arguments, respectively, in VS FORTRAN.

[2] $\gamma = 16^{63} (1 - 16^{-6})$ for regular precision routines, $16^{63} (1 - 16^{-16})$ for double-precision routines, and $16^{63} (1 - 16^{-28})$ for extended precision.

[3] All functions are generated as out-of-line library calls.

Figure 23. Hyperbolic Functions

General Function	Entry Name	Definition	Argument(s) No.	Argument(s) Type[1]	Argument(s) Range	Function Value Type[1] and Range[5]	Error Code
Absolute value	IABS	$y = \|x\|$	1	INTEGER *4	any INTEGER argument	INTEGER *4 $0 \leqq y \leqq \gamma$	None. See Note 9.
	ABS	$y = \|x\|$	1	REAL *4	any REAL argument	REAL *4 $0 \leqq y \leqq \gamma$	None. See Note 9.
	DABS	$y = \|x\|$	1	REAL *8	any REAL argument	REAL *8 $0 \leqq y \leqq \gamma$	None. See Note 9.
	QABS	$y = \|x\|$	1	REAL *16	any REAL argument	REAL *16 $0 \leqq y \leqq \gamma$	None. See Note 9.
	CABS	$y=\|z\|= (x_1^2 + x_2^2)^{1/2}$	1	COMPLEX *8	any COMPLEX argument See Note 2	REAL *4 $0 \leqq y_1 \leqq \gamma$ $y_2 = 0$	None. See Note 10.
	CDABS	$y=\|z\|= (x_1^2 + x_2^2)^{1/2}$	1	COMPLEX *16	any COMPLEX argument See Note 2	REAL *8 $0 \leqq y_1 \leqq \gamma$ $y_2 = 0$	None. See Note 10.
	CQABS	$y = \|z\| = (x_1^2 + x_2^2)^{1/2}$	1	COMPLEX *32	any COMPLEX argument See Note 2	REAL *16 $0 \leqq y_1 \leqq \gamma$ $y_2 = 0$	None. See Note 10.
Error function	ERF	$y=\frac{2}{\sqrt{\pi}}\int_0^x e^{-u^2} du$	1	REAL *4	any REAL argument	REAL *4 $-1 \leqq y \leqq 1$	None. See Note 10.
	ERFC	$y=\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-u^2} du$ $y=1 - erf(x)$	1	REAL *4	any REAL argument	REAL *4 $0 \leqq y \leqq 2$	None. See Note 10.
	DERF	$y=\frac{2}{\sqrt{\pi}}\int_0^x e^{-u^2} du$	1	REAL *8	any REAL argument	REAL *8 $-1 \leqq y \leqq 1$	None. See Note 10.
	DERFC	$y=\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-u^2} du$ $y=1 - erf(x)$	1	REAL *8	any REAL argument	REAL *8 $0 \leqq y \leqq 2$	None. See Note 10.
	QERF	$y=\frac{2}{\sqrt{\pi}}\int_0^x e^{-u^2} du$	1	REAL *16	any REAL argument	REAL *16 $-1 \leqq y \leqq 1$	None. See Note 10.
	QERFC	$y=\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-u^2} du$ $y=1 - erf(x)$	1	REAL *16	any REAL argument	REAL *16 $0 \leqq y \leqq 2$	None. See Note 10.

NOTES: (See end of figure.)

Figure 24 (Part 1 of 4). Miscellaneous Mathematical Functions

General Function	Entry Name	Definition	Argument(s)			Function Value Type[1] and Range[5]	Error Code
			No.	Type[1]	Range		
Gamma and log-gamma	GAMMA	$y=\int_0^\infty u^{x-1} e^{-u} du$	1	REAL *4	$x > 2^{-252}$ and $x < 57.5744$	REAL *4 $0.88560 \leqq y \leqq \gamma$	290 See Note 10.
	ALGAMA	$y=\log_e \Gamma(x)$ or $y=\log_e \int_0^\infty u^{x-1} e^{-u} du$	1	REAL *4	$x > 0$ and $x < 4.2913 \cdot 10^{73}$	REAL *4 $-0.12149 \leqq y \leqq \gamma$	291 See Note 10.
	DGAMMA	$y=\int_0^\infty u^{x-1} e^{-u} du$	1	REAL *8	$x > 2^{-252}$ and $x < 57.5744$	REAL *8 $0.88560 \leqq y \leqq \gamma$	300 See Note 10.
	DLGAMA	$y=\log_e \Gamma(x)$ or $y=\log_e \int_0^\infty u^{x-1} e^{-u} du$	1	REAL *8	$x > 0$ and $x < 4.2913 \cdot 10^{73}$	REAL *8 $-0.12149 \leqq y \leqq \gamma$	301 See Note 10.
Square root	SQRT	$y=\sqrt{x}$ or $y=x^{1/2}$	1	REAL *4	$x \geqq 0$	REAL *4 $0 \leqq y \leqq \gamma^{1/2}$	251 See Note 10.
	DSQRT	$y=\sqrt{x}$ or $y=x^{1/2}$	1	REAL *8	$x \geqq 0$	REAL *8 $0 \leqq y \leqq \gamma^{1/2}$	261 See Note 10.
	CSQRT	$y=\sqrt{z}$ or $y=z^{1/2}$ See Note 7	1	COMPLEX *8	any COMPLEX argument	COMPLEX *8 $0 \leqq y_1 \leqq 1.0987(\gamma^{1/2})$ $\lvert y_2 \rvert \leqq 1.0987(\gamma^{1/2})$	None. See Note 10.
	CDSQRT	$y=\sqrt{z}$ or $y=z^{1/2}$ See Note 7	1	COMPLEX *16	any COMPLEX argument	COMPLEX *16 $0 \leqq y_1 \leqq 1.0987(\gamma^{1/2})$ $\lvert y_2 \rvert \leqq 1.0987(\gamma^{1/2})$	None. See Note 10.
	QSQRT	$y = \sqrt{x}$ or $y = x^{1/2}$	1	REAL *16	$x \geqq 0$	REAL *16 $0 \leqq y \leqq \gamma^{1/2}$	289 See Note 10.
	CQSQRT	$y = \sqrt{z}$ or $y = z^{1/2}$ See Note 7	1	COMPLEX *32	any COMPLEX argument	COMPLEX *32 $0 \leqq y_1 \leqq 1.0987(\gamma^{1/2})$ $y_2 \leqq 1.0987(\gamma^{1/2})$	None. See Note 10.
NOTES: (See end of figure.)							

Figure 24 (Part 2 of 4). Miscellaneous Mathematical Functions

General Function	Entry Name	Definition	Argument(s)			Function Value Type[1] and Range[5]	Error Code
			No.	Type[1]	Range		
Modular arithmetic	MOD	$y=x_1$ (modulo x_2) See Note 3	2	INTEGER	$x_2 \neq 0$ See Note 4	INTEGER *4	None. See Note 9.
	AMOD		2	REAL *4	$x_2 \neq 0$ See Note 4	REAL *4	None. See Note 9.
	DMOD		2	REAL *8	$x_2 \neq 0$ See Note 4	REAL *8	None. See Note 9.
	QMOD		2	REAL *16	$x_2 \neq 0$ See Note 4	REAL *16	None. See Note 9.
Truncation	AINT	y = (sign of x)•n where $n = \lfloor \lvert x \rvert \rfloor$ See Note 6	1	REAL *4	any REAL argument	REAL *4	None. See Note 9.
	DINT		1	REAL *8	any REAL argument	REAL *8	None. See Note 9.
	QINT		1	REAL *16	any REAL argument	REAL *16	None. See Note 9.
Obtain imaginary part of a complex argument	AIMAG		1	COMPLEX *8	any COMPLEX argument	REAL *4	None. See Note 9.
	DIMAG		1	COMPLEX *16	any COMPLEX argument	REAL *8	None. See Note 9.
	QIMAG		1	COMPLEX *32	any COMPLEX argument	REAL *16	None. See Note 9.
Obtain conjugate of a complex argument	CONJG	$y = x_1 - x_2 i$ for argument = $x_1 + x_2 i$	1	COMPLEX *8	any COMPLEX argument	COMPLEX *8	None. See Note 9.
	DCONJG		1	COMPLEX *16	any COMPLEX argument	COMPLEX *16	None. See Note 9.
	QCONJG		1	COMPLEX *32	any COMPLEX argument	COMPLEX *32	None. See Note 9.
NOTES: (See end of figure.)							

Figure 24 (Part 3 of 4). Miscellaneous Mathematical Functions

General Function	Entry Name	Definition	Argument(s)			Function Value Type[1] and Range[5]	Error Code
			No.	Type[1]	Range		
Nearest whole number	ANINT	$y = (\text{sign of } x) \cdot v$ where $v = [\,\lvert x + .5 \rvert\,]$ if $x \geq 0$ or	1	REAL *4	any REAL argument	REAL *4	None. See Note 9.
	DNINT	$v = [\,\lvert x - .5 \rvert\,]$ if $x < 0$. See Note 6	1	REAL *8	any REAL argument	REAL *8	None. See Note 9.
Nearest integer	NINT	$y = (\text{sign of } x) \cdot n$ where $n = [\,\lvert x + .5 \rvert\,]$ if $x \geq 0$ or	1	REAL *4	any REAL argument	INTEGER *4	None. See Note 9.
	IDNINT	$n = [\,\lvert x - .5 \rvert\,]$ if $x < 0$. See Note 8	1	REAL *4	any REAL*8 argument	INTEGER *4	None. See Note 9.
Positive difference	IDIM	$y = x_1 - x_2$ if $x_1 > x_2$ $y = 0$ if $x_1 \leq x_2$	2	INTEGER *4	any INTEGER argument	INTEGER *4	None. See Note 9.
	DIM		2	REAL *4	any REAL argument	REAL *4	None. See Note 9.
	DDIM		2	REAL *8		REAL *8	None. See Note 9.
	QDIM		2	REAL *16		REAL *16	None. See Note 9.
Transfer of sign	ISIGN	$y = \lvert x_1 \rvert$ if $x_2 \geq 0$ $y = -\lvert x_1 \rvert$ if $x_2 < 0$	2	INTEGER *4	any INTEGER argument	INTEGER *4	None. See Note 9.
	SIGN		2	REAL *4	any REAL argument	REAL *4	None. See Note 9.
	DSIGN		2	REAL *8		REAL *8	None. See Note 9.
	QSIGN		2	REAL *16		REAL *16	None. See Note 9.
Double precision product	DPROD	$y = x_1 * x_2$	2	REAL *4	any REAL argument	REAL *8	None. See Note 9.

NOTES:

[1] REAL *4, REAL *8, and REAL *16 arguments correspond to REAL, DOUBLE PRECISION, and EXTENDED PRECISION arguments, respectively, in VS FORTRAN.

[2] Floating-point overflow can occur.

[3] The expression x_1 (modulo x_2) is defined as $x_1 - \left[\frac{x_1}{x_2}\right] \cdot x_2$, where the brackets indicate that an integer is used. The largest integer whose magnitude does not exceed the magnitude of $\frac{x_1}{x_2}$ is used. The sign of the integer is the same as the sign of $\frac{x_1}{x_2}$.

[4] If $x_2 = 0$, then the modulus function is mathematically undefined. In addition, a divide exception is recognized and an interruption occurs. (A detailed description of the interruption procedure is given in Appendix C.)

[5] $\gamma = 16^{63}(1 - 16^{-6})$ for regular precision routines, $16^{63}(1 - 16^{-14})$ for double-precision routines and $16^{63}(1 - 16^{-28})$ for extended precision routines.

[6] $[\,\lvert x \rvert\,]$ is such that $v = \lvert m \rvert$ where m is the greatest integer satisfying the relationship $\lvert m \rvert \leq \lvert x \rvert$, and the resulting v is expressed as a real value.

[7] z is a complex-number of the form $x_1 + x_2 i$.

[8] $[\,\lvert x \rvert\,]$ is such that $n = \lvert m \rvert$ where m is the greatest integer satisfying the relationship $\lvert m \rvert \leq \lvert x \rvert$.

[9] This function is generated inline.

[10] This function is generated as an out-of-line library call.

Figure 24 (Part 4 of 4). Miscellaneous Mathematical Functions

General Function	Generic Name	Entry Name[5]	Definition	Argument(s) No.	Argument(s) Type	Argument(s) Range	Function Value Type and Range
Conversion to integer	INT	See Note 1	y = (sign of x) • n where n is the largest integer ≤ |x|	1	INTEGER *4	any INTEGER argument	INTEGER *4
		INT See Note 2		1	REAL *4	any REAL argument	INTEGER *4
		IDINT		1	REAL *8	any REAL argument	INTEGER *4
		IQINT		1	REAL *16	any REAL argument	INTEGER *4
		See Note 1	for $z = x_1 + x_2 i$, $y = INT(x_1)$	1	COMPLEX *8	any COMPLEX argument	INTEGER *4
	See Note 3	HFIX	y = (sign of x) • n where n is the largest integer ≤ |x|	1	REAL *4	any REAL argument	INTEGER *2
Conversion to real	REAL	REAL See Note 4		1	INTEGER *4	any INTEGER argument	REAL *4
		See Note 1		1	REAL *4	any REAL argument	REAL *4
		SNGL		1	REAL *8	any REAL argument	REAL *4
		SNGLQ		1	REAL *16	any REAL argument	REAL *4
		See Note 1	for $z = x_1 + x_2 i$, $y = REAL(x_1)$	1	COMPLEX *8	any COMPLEX argument	REAL *4
		DREAL		1	COMPLEX *16	any COMPLEX argument	REAL *8
		QREAL		1	COMPLEX *32	any COMPLEX argument	REAL *16
Conversion to double	DBLE	DFLOAT		1	INTEGER *4	any INTEGER argument	REAL *8
		DBLE		1	REAL *4	any REAL argument	REAL *8
		See Note 1		1	REAL *8	any REAL argument	REAL *8
		DBLEQ		1	REAL *16	any REAL argument	REAL *8
		See Note 1	for $z = x_1 + x_2 i$, $y = DBLE (x_1)$	1	COMPLEX *8	any COMPLEX argument	REAL *8

NOTES: (See end of figure.)

Figure 25 (Part 1 of 3). Conversion and Maximum/Minimum Functions

General Function	Generic Name	Entry Name	Definition	Argument(s)			Function Value Type and Range
				No.	Type	Range	
Conversion to extended precision	QEXT	QFLOAT		1	INTEGER*4	any INTEGER argument	REAL*16
		QEXT		1	REAL*4	any REAL argument	REAL*16
		QEXTD		1	REAL*8	any REAL argument	REAL*16
Conversion to complex	CMPLX	See Note 1	$y = x_1 + x_2i$ where x_1 = REAL(arg) and $x_2 = 0.$	1	INTEGER*4	any INTEGER argument	COMPLEX*8
		CMPLX		1	REAL*4	any REAL argument	COMPLEX*8
		See Note 1		1	REAL*8	any REAL argument	COMPLEX*8
		QCMPLX	$y = x_1 + x_2i$ where x_1 = arg and x_2 = 0.Q0	1	REAL*16	any REAL argument	COMPLEX*32
		See Note 1	$y = x_1 + x_2i$ for arg = $x_1 + x_2i$	1	COMPLEX*8	any COMPLEX argument	COMPLEX*8
	See Note 3	DCMPLX	$y = x_1 + x_2i$ where x_1 = arg and $x_2 = 0.$	1	REAL*8	any REAL argument	COMPLEX*16
	CMPLX	See Note 1	$y = x_1 + x_2i$ where x_1 = REAL(arg1) and x_2 = REAL(arg2)	2	INTEGER*4	any INTEGER argument	COMPLEX*8
		CMPLX		2	REAL*4	any REAL argument	COMPLEX*8
		See Note 1		2	REAL*8	any REAL argument	COMPLEX*8
		QCMPLX	$y = x_1 + x_2i$ where x_1 = arg1 and x_2 = arg2	2	REAL*16	any REAL argument	COMPLEX*32
	See Note 3	DCMPLX	$y = x_1 + x_2i$ where x_1 = arg1 and x_2 = arg2	2	REAL*8	any REAL argument	COMPLEX*16
NOTES: (See end of figure.)							

Figure 25 (Part 2 of 3). Conversion and Maximum/Minimum Functions

General Function	Generic Name	Entry Name	Definition	Argument(s)			Function Value Type and Range
				No.	Type	Range	
Maximum value	MAX	MAX0	$y = \max(x_1, \ldots x_n)$	⩾ 2	INTEGER*4	any INTEGER arguments	INTEGER*4
		AMAX1		⩾ 2	REAL*4	any REAL arguments	REAL*4
		DMAX1		⩾ 2	REAL*8	any REAL arguments	REAL*8
		QMAXI		⩾ 2	REAL*16	any REAL arguments	REAL*16
	See Note 3	AMAX0		⩾ 2	INTEGER*4	any INTEGER arguments	REAL*4
	See Note 3	MAX1		⩾ 2	REAL*4	any REAL arguments	INTEGER*4
Minimum value	MIN	MIN0	$y = \min(x_1, \ldots x_n)$	⩾ 2	INTEGER*4	any INTEGER arguments	INTEGER*4
		AMIN1		⩾ 2	REAL*4	any REAL arguments	REAL*4
		DMIN1		⩾ 2	REAL*8	any REAL arguments	REAL*8
		QMIN1		⩾ 2	REAL*16	any REAL arguments	REAL*16
	See Note 3	AMIN0		⩾ 2	INTEGER*4	any INTEGER arguments	REAL*4
	See Note 3	MIN1		⩾ 2	REAL*4	any REAL arguments	INTEGER*4

NOTES:

[1] No specific name exists for this case. The generic name must be used for this argument type.

[2] IFIX is an alternate specific name for this function.

[3] Specific name must be used to obtain function value of this type.

[4] FLOAT is an alternate specific name for this function.

[5] All functions in all parts of this figure are inline functions. None of the function names can be passed as arguments. There are no library error codes because there are no library routines.

Figure 25 (Part 3 of 3). Conversion and Maximum/Minimum Functions

General Function[2]	Entry Name	Definition	Argument(s)		Function Value Type and Range	Error Code
			No.	Type		
Convert character to integer	ICHAR	Position of character in EBCDIC collating sequence	1	CHARACTER	INTEGER *4	None
Convert integer to character	CHAR	Character corresponding to position of argument in EBCDIC collating sequence	1	INTEGER *4	CHARACTER *1	188
Length of character item	LEN	Length of character entity	1	CHARACTER	INTEGER *4	None
Index of character item	INDEX	Location of substring a_2 in string a_1	2	CHARACTER	INTEGER *4	189, 190
Alphamerically greater than or equal	LGE	$a_1 \geq a_2$ See Note 1	2	CHARACTER	LOGICAL *4	191, 192
Alphamerically greater than	LGT	$a_1 > a_2$ See Note 1	2	CHARACTER	LOGICAL *4	191, 192
Alphamerically less than or equal	LLE	$a_1 \leq a_2$ See Note 1	2	CHARACTER	LOGICAL *4	191, 192
Alphamerically less than	LLT	$a_1 < a_2$ See Note 1	2	CHARACTER	LOGICAL *4	191, 192

NOTES:

[1] Comparison is made using the ASCII collating sequence.

[2] All functions are generated as out-of-line library calls.

Figure 26. Character Manipulation Functions

General Function	Entry Name	Definition	Argument(s)			Function Value Type and Range	Error Code
			No.	Type	Range		
Logical AND	IAND	k = and (i,j)	2	INTEGER *4	any INTEGER arguments	INTEGER *4	None
Logical OR	IOR	k = or (i,j)	2	INTEGER *4	any INTEGER arguments	INTEGER *4	None
Logical exclusive OR	IEOR	k = xor (i,j)	2	INTEGER *4	any INTEGER arguments	INTEGER *4	None
Logical complement	NOT	k = not (i)	1	INTEGER *4	any INTEGER argument	INTEGER *4	None
Shift operation	ISHFT	k = shift (i,m) i is shifted by m where for $m < 0$, shift is right; $m > 0$, shift is left; and $m = 0$, no shift	2	INTEGER *4	i is any INTEGER argument; $-32 \leq m \leq 32$	INTEGER *4	159
Bit testing and setting	BTEST	1 = bittest (i,m) tests m-th bit of argument i	2	INTEGER *4	i is any INTEGER argument; $0 \leq m \leq 31$ See Note 3	LOGICAL *4	159
	IBSET	k = bitset (i,m) sets m-th bit of argument i to 1:	2	INTEGER *4		INTEGER *4	159
	IBCLR	k = bitclear (i,m) sets m-th bit of argument i to 0.	2	INTEGER *4		INTEGER *4	159

NOTES:

[1] There are no generic names for the bit manipulation functions. All specific names may be passed as actual arguments.

[2] The first four functions are always inline. The second four are inline if the second argument is an integer constant; a library function is called if the second argument is an integer variable or expression.

[3] The bits in the first argument (*i*) are numbered from right to left, beginning at zero. Thus $m = 0$ corresponds to the rightmost bit of the argument *i*.

Figure 27. Bit Manipulation Functions

Generic Name	Definition	Number, Type, and Length of Arguments[1,4]								Function Value[2]	
		No.	I*4	R*4	R*8	R*16	C*8	C*16	C*32	Type	Length
ABS	Absolute value	1	X	X	X	X				Argument	Argument
							X	X	X	Real	1/2 Argument
ACOS	Arc cosine	1		X	X	X				Real	Argument
AINT	Truncation	1		X	X	X				Real	Argument
ANINT	Nearest whole number	1		X	X					Real	Argument
ASIN	Arc sine	1		X	X	X				Real	Argument
ATAN	Arc tangent	1		X	X	X				Real	Argument
ATAN2	Arc tangent (2 arguments)	2		X	X	X				Real	Argument
CMPLX	Conversion to complex	1	X	X	X		X			Complex	8
	(See Note 3.)	1				X				Complex	32
		2	X	X	X					Complex	8
		2				X				Complex	32
CONJG	Conjugate	1					X	X	X	Complex	Argument
COS	Cosine	1		X	X	X	X	X	X	Argument	Argument
COSH	Hyperbolic cosine	1		X	X	X				Real	Argument
COTAN	Cotangent	1		X	X	X				Real	Argument
DBLE	Express as R*8	1	X	X	X	X	X			Real	8
DIM	Positive difference	2	X	X	X	X				Argument	Argument
ERF	Error function	1		X	X	X				Real	Argument
ERFC	1 – Error function	1		X	X	X				Real	Argument
EXP	Exponentiation	1		X	X	X	X	X	X	Argument	Argument
GAMMA	Gamma function	1		X	X					Real	Argument
IMAG	Imaginary part	1					X	X	X	Real	1/2 Argument
INT	Express as I*4	1	X	X	X	X	X			Integer	4
LGAMMA	Log of gamma function	1		X	X					Real	Argument
LOG	Natural logarithm	1		X	X	X	X	X	X	Argument	Argument
LOG10	Common logarithm	1		X	X	X				Real	Argument
MAX	Maximum value	≥ 2	X	X	X	X				Argument	Argument
MIN	Minimum value	≥ 2	X	X	X	X				Argument	Argument
MOD	Remainder	2	X	X	X	X				Argument	Argument
NINT	Nearest integer	1		X	X					Integer	4
QEXT	Express as R*16	1	X	X	X					Real	16
REAL	Conversion to real	1	X	X	X	X				Real	4
		1					X	X	X	Real	1/2 Argument
SIGN	Transfer of sign	2	X	X	X	X				Argument	Argument
SIN	Sine	1		X	X	X	X	X	X	Argument	Argument
SINH	Hyperbolic sine	1		X	X	X				Real	Argument
SQRT	Square root	1		X	X	X	X	X	X	Argument	Argument
TAN	Tangent	1		X	X	X				Real	Argument
TANH	Hyperbolic tangent	1		X	X	X				Real	Argument

NOTES:

[1] "X" indicates a permissible mode of argument.

[2] "Argument" indicates that the type or length of the result is the same as that of the argument(s).

[3] The specific name DCMPLX must be used to convert an R*8 argument to a C*16 value (or to convert and express two R*8 arguments as a C*16 value.)

[4] If more than one argument is permitted, all arguments must be of same type and length.

Figure 28. Generic Names for Intrinsic Functions

Part 2. Library Reference

The following topics are discussed in Part 2:

Introduction

Mathematical, Character, and Bit Subprograms

Service and Utility Subroutine Subprograms

Extended Error Handling Subroutines and Error Option Table

See Appendix I, "Library Procedures and Messages" on page 463 for related error messages.

Chapter 7. Introduction

The VS FORTRAN library contains the following categories of subprograms:

- Input/output operation subprograms
- Data conversion subprograms
- Mathematical, character, and bit subprograms
- Service and utility subroutine subprograms
- Extended error handling subprograms
- Initialization and termination subprograms

The input/output operation and data conversion subprograms are accessed for you, via compiler-generated calls, in response to REWRITE, DELETE, REWIND, BACKSPACE, ENDFILE, INQUIRE, PRINT, WAIT, OPEN, CLOSE, READ, WRITE, and FORMAT statements. Usage information for these statements, and thus for these subprograms, is in Chapter 5, "VS FORTRAN Statements," and in *VS FORTRAN Programming Guide*.

The mathematical, character, and bit subprograms make up the intrinsic functions, which you refer to directly by name, and the notational functions, which are accessed for you in response to mathematical notation. Usage information for these subprograms is in Chapter 6, "VS FORTRAN Intrinsic Functions," and Chapter 8, "Mathematical, Character, and Bit Subprograms."

The service and utility subprograms (such as checking mathematical exceptions and dumping data areas) are called by you directly. Usage information for these subprograms is in Chapter 9, "Service and Utility Subroutines."

The extended error handling subprograms enable you to provide user error exits and change error handling operations. You call these subprograms directly. Usage information is in Chapter 10, "Extended Error Handling Subroutines and Error Option Table."

The initialization and termination subprograms are accessed for you, via compiler-generated calls, in the main VS FORTRAN program. There is no usage information for these subprograms because you are not able to control them. However, if you write subprograms in VS FORTRAN, or if you call VS FORTRAN library subprograms from a non-VS FORTRAN main program, the initialization and termination subprograms usually must be called at the beginning and end of the main program. For more information, see Appendix G, "Assembler Language Information."

Chapter 8. Mathematical, Character, and Bit Subprograms

The mathematical, character, and bit subprograms supplied in the VS FORTRAN library perform commonly used computations and conversions. These subprograms are called in two ways:

- Explicitly, when the appropriate name appears in a source language statement, or
- Implicitly, when certain notation appears in a source language statement

The material in this chapter describes explicitly called routines under "Explicitly Called Subprograms" (below) and implicitly called routines under "Implicitly Called Subprograms" on page 308.

Accuracy statistics are given in Appendix F, "Accuracy Statistics" on page 425; storage estimates appear in Appendix E, "Storage Estimates" on page 417.

Detailed information for calling the subprograms from assembler language is given in Appendix G, "Assembler Language Information" on page 433.

For a small subset of the standard mathematical subprograms, alternative subprograms are available that provide more accurate results with arguments of large absolute value, and in certain instances, provide faster computation. These routines are referred to as the *alternative mathematical library* in VS FORTRAN. Alternative routines are available for the intrinsic functions DSIN, DCOS, DTAN, DCOTAN, EXP, and DEXP, and for the implicitly called functions FDXPD# and FRXPR#. *VS FORTRAN Compiler and Library Installation and Customization* describes how these routines can be installed for your use.

Explicitly Called Subprograms

All the explicitly called subprograms are VS FORTRAN intrinsic functions. Each of these functions performs a mathematical, character, or bit manipulation. See Chapter 6, "VS FORTRAN Intrinsic Functions" on page 285 for detailed information about these functions. See Appendix D, "Algorithms for Library Mathematical Functions" on page 369, for information about the algorithms used in many of them.

Implicitly Called Subprograms

The implicitly called subprograms are executed as a result of certain notation appearing in a VS FORTRAN source statement. The VS FORTRAN compiler generates the instructions necessary to call the appropriate subprogram. For example, for the following source statement:

```
ANS = BASE**EXPON
```

where BASE and EXPON are REAL*4 variables, the VS FORTRAN compiler generates a reference to FRXPR#, the entry name for a subprogram that raises a real number to a real power.

The implicitly called mathematical and character subprograms in the VS FORTRAN library are described in Figure 29 on page 309 and Figure 30 on page 310. The column headed "Implicit Function Reference" shows a representation of a source statement that might appear in a VS FORTRAN source module and cause the subprogram to be called. The rest of the column headings have the same meaning as those used with the explicitly called subprograms. Implicitly called service subprograms are in Figure 31 on page 311.

For subprograms that involve exponentiation, the action taken within a subprogram depends upon the types of the base and exponent used. Figure 32 on page 311 through Figure 35 on page 312 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.

General Function	Entry[1] Name	Implicit Function Reference[2]	Argument(s) No.	Argument(s) Type[3]	Function Value Type[3]	Error Code
Multiply and divide complex numbers	CDMPY#	$y = z_1 * z_2$	2	COMPLEX *16	COMPLEX *16	
	CDDVD#	$y = z_1/z_2$	2	COMPLEX *16	COMPLEX *16	
	CMPY#	$y = z_1 * z_2$	2	COMPLEX *8	COMPLEX *8	
	CDVD#	$y = z_1/z_2$	2	COMPLEX *8	COMPLEX *8	
	CQMPY#	$y = z_1 * z_2$	2	COMPLEX *32	COMPLEX *32	
	CQDVD#	$y = z_1/z_2$	2	COMPLEX *32	COMPLEX *32	
Compare of complex numbers	CXMPR# See Note 4	$y = z_1$ compop z_2 See Note 5	2	COMPLEX (of all lengths)	LOGICAL*4	
Raise an integer to an integer power	FIXPI#	y = i**j	2	i = INTEGER *4 j = INTEGER *4	INTEGER *4	241
Raise a real number to an integer power	FRXPI#	y = a**j	2	a = REAL *4 j = INTEGER *4	REAL *4	242
	FDXPI#	y = a**j	2	a = REAL *8 j = INTEGER *4	REAL *8	243
	FQXPI#	y = a**j	2	a = REAL *16 j = INTEGER *4	REAL*16	248
Raise a real number to a real power	FRXPR# See Note 6	y = a**b	2	a = REAL *4 b = REAL *4	REAL *4	244
	FDXPD# See Note 6	y = a**b	2	a = REAL *8 b = REAL *8	REAL *8	245
	FQXPQ#	y = a**b	2	a = REAL *16 b = REAL *16	REAL *16	249, 250
Raise 2 to a real power	FQXP2#	y = 2**b	1	b = REAL *16	REAL *16	260
Raise a complex number to an integer power	FCDXI#	y = z**j	2	z = COMPLEX *16 j = INTEGER *4	COMPLEX *16	247
	FCXPI#	y = z**j	2	z = COMPLEX *8 j = INTEGER *4	COMPLEX *8	246
	FCQXI#	y = z**j	2	z = COMPLEX *32 j = INTEGER *4	COMPLEX *32	270

Figure 29. Implicitly Called Mathematical Subprograms

Notes to Figure 29:

1. This name must be used in an assembler language program to call the subprogram; 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 subprogram may be called.

3. REAL*4, REAL*8, and REAL*16 arguments correspond to real, double precision, and extended precision arguments, respectively, in VS FORTRAN.

4. CXMPR# is an entry name in the library module IFYCCMPR, which is also used for a compare of character arguments.

5. *compop* is one of the following relational operators: equal or not equal.

6. Available also in the alternative mathematical library.

Entry Name	Implicit Function Reference	Argument(s)		Function Value Type	Error Code
		No.	Type		
CCMPR# See Note 2	$y = x_1$ compop x_2 See Note 1	6	Character	Any character argument	193 194
CMOVE# See Note 2	$y = x$	4	Character	Any character argument	195 196 197
CNCAT#	$y = x_1 // x_2 \ldots // x_n$	≧2	Character	Any character argument	198 199

Figure 30. Implicitly Called Character Subprograms

Notes to Figure 30:

1. Where *compop* is one of the following relational operators:

   equal
   not equal
   greater than
   less than
   greater than or equal
   less than or equal

   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 Compiler, the library functions used for the comparison of character type items and for the assignment of character type items are not invoked. All these operations are performed inline. These routines remain in the VS FORTRAN Library to support programs compiled with releases of the compiler earlier than Release 4.0.

Entry Name	Function	Arguments	Error Code
DSPAN # DSPN2 # DSPN4 #	Calculate dimension factors and span of adjustable dimension array.	Array description	187
DYCMN#	Obtain storage and relocate adcons for DYNAMIC COMMON.	COMMON and adcon information	156 157 158

Figure 31. Implicitly Called Service Subprograms

Base (I)	Exponent (J)		
	J > 0	J = 0	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 32. Exponentiation with Integer Base and Exponent

Base (A)	Exponent (J)		
	J > 0	J = 0	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 33. Exponentiation with Real Base and Integer Exponent

Base (A)	Exponent (B)		
	B > 0	B = 0	B < 0
A > 0	Compute the function value	Function value = 1	Compute the function value
A = 0	Function value = 0	Error message 244 or 245	Error message 244 or 245
A < 0	Error message 253 or 263	Function value = 1	Error message 253 or 263

Figure 34. Exponentiation with Real Base and Exponent

Base (C) C = P + Qi	Exponent (J)		
	J > 0	J = 0	J < 0
P > 0 and Q > 0	Compute the function value	Function value=1 + 0*i*	Compute the function value
P > 0 and Q = 0	Compute the function value	Function value=1 + 0*i*	Compute the function value
P > 0 and Q < 0	Compute the function value	Function value=1 + 0*i*	Compute the function value
P = 0 and Q > 0	Compute the function value	Function value=1 + 0*i*	Compute the function value
P = 0 and Q = 0	Function value 0 + 0*i*	Error message 246 or 247	Error message 246 or 247
P = 0 and Q < 0	Compute the function value	Function value=1 + 0*i*	Compute the function value
P < 0 and Q > 0	Compute the function value	Function value=1 + 0*i*	Compute the function value
P < 0 and Q = 0	Compute the function value	Function value=1 + 0*i*	Compute the function value
P < 0 and Q < 0	Compute the function value	Function value=1 + 0*i*	Compute the function value

Figure 35. Exponentiation with Complex Base and Integer Exponent

Chapter 9. Service and Utility Subroutines

The service and utility subroutines applied in the VS FORTRAN library perform tests for mathematical exceptions and utility functions, respectively. The subroutines are called by the appropriate entry name in a VS FORTRAN language CALL statement.

Mathematical Exception Test Subprograms

These subprograms test the status of indicators and may return a value to the calling program. In the following description of the subprograms, *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

An integer or real variable in the program unit.

The values of *k* returned have the following meanings:

Value	Meaning
1	The divide-check indicator is *on*.
2	The divide-check indicator is *off*.

OVERFL Subroutine

The OVERFL subroutine tests for exponent overflow or underflow, and returns a value indicating the existing condition. After testing, the overflow indicator is turned off.

Syntax

CALL OVERFL (*k*)

k

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).

Utility Subprograms

The utility subprograms perform service operations for the VS FORTRAN programmer; for example,

- Dump a specified area of storage: DUMP/PDUMP.
- Dump a specified area of storage containing character data: CDUMP/CPDUMP.
- Terminate execution: EXIT.
- Load a phase and modify the unit assignment table: OPSYS (VSE only).
- Provide a symbolic dump of all variables in a program unit: SDUMP.
- Allow or suppress a program interrupt due to exponent underflow: XUFLOW

DUMP/PDUMP Subroutine

The DUMP/PDUMP subroutine dynamically dumps a specified area of storage onto the system output data set. After the dump, for entry, DUMP execution is terminated, or, for entry, PDUMP execution is continued.

Syntax

CALL {DUMP | PDUMP} (*a*1,*b*1,*k*1,...*a*n,*b*n,*k*n)

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 limits of the storage area.

k

Specifies the dump format to be used.

The values that can be specified for *k* and their meanings are: :ihi see='utility subprograms'.storage dump

Value	Format Requested
0	Hexadecimal
1	LOGICAL*1
2	LOGICAL*4
3	INTEGER*2
4	INTEGER*4
5	REAL*4
6	REAL*8
7	COMPLEX*8
8	COMPLEX*16
9	CHARACTER
10	REAL*16
11	COMPLEX*32

Programming Considerations for DUMP/PDUMP

A load module or phase may occupy a different area of storage each time it is executed. To ensure that the appropriate areas of storage are dumped, the following conventions should be observed.

If an array and a variable are to be dumped at the same time, a separate set of arguments should be used for the array and for the variable. The specification of limits for the array should be from the first element in the array to the last element. For example, assume that A is a variable in common, B is a real number, and TABLE is an array of 20 elements. The following call to the storage dump subprogram could be used to dump TABLE and B in hexadecimal format, and terminate execution 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 subprogram could be used to dump the variables A and B in REAL*8 format without terminating execution:

```
CALL PDUMP(A,A,6,B,B,6)
```

If variables not in common are to be dumped, each variable must be listed separately in the argument list. For example, if R, P, and Q are defined implicitly in the program, the statement

```
CALL PDUMP(R,R,5,P,P,5,Q,Q,5)
```

should be used to dump the three variables in REAL*4 format. If the statement

```
CALL PDUMP(R,Q,5)
```

is used, all main storage between R and Q is dumped, which may or may not include P, and may include other variables.

If an array and a variable are passed to a subroutine as arguments, the arguments in the call to the storage dump subprogram in the subroutine should specify the parameters used in the definition of the subroutine. For example, if the subroutine SUBI is defined as:

```
SUBROUTINE SUBI(C,Y)
DIMENSION X (10)
```

and the call to SUBI within the source module is:

```
DIMENSION A (10)
      .
      .
      .
CALL SUBI (A,B)
```

then the following statement should be used in SUBI to dump the variables in hexadecimal format without terminating execution:

```
CALL PDUMP (X(1),X(10),0,Y,Y,0)
```

If the statement

```
CALL PDUMP (X(1),Y,0)
```

is used, all storage between (1) and Y is dumped because of the method of transmitting arguments.

When hexadecimal (0) or literal (9) is specified, the programmer should realize that the upper limit is assumed to be of length 4.

CDUMP/CPDUMP Subroutine

The CDUMP/CPDUMP subroutine dynamically dumps a specified area of storage containing character data. After the dump, for entry, CDUMP execution is terminated, or for entry, CPDUMP execution is continued.

Syntax

CALL {CDUMP | CPDUMP} (*a*1,*b*1,...,*a*n,*b*n)

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 limits of each storage area.

The dump is always produced in character format. (A dump format type (as for DUMP/PDUMP) must not be specified.)

Programming Considerations for CDUMP/CPDUMP

A load module may occupy a different area of storage each time it is executed. To ensure that the appropriate areas of storage are dumped, the following conventions should be observed.

If an array and a variable are to be dumped at the same time, a separate set of arguments should be used for the array and for the variable. The specification of limits for the array should be from the first element in the array to the last element. For example, assume that A is a variable in common, B is a real number, and TABLE is an array of 20 elements. The following call to the storage dump subprogram could be used to dump TABLE and B in hexadecimal format, and terminate execution after the dump is taken:

CALL CDUMP(a_1 ,b_1 ,...,a_n ,b_n)

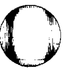
EXIT Subroutine

The EXIT subroutine terminates execution of the load module or phase 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 can be produced.



OPSYS Subroutine (VSE Only)

The OPSYS subroutine has two forms.

1. CALL OPSYS statement to run multiphase jobs:

Syntax

CALL OPSYS('LOAD','*phasename*')

LOAD
The OPSYS parameter specifying this function.

phasename
Specifies the name of the phase to be loaded. The phase must be in the core image library.

The '*phasename*' must be specified in eight alphameric characters. If fewer than eight characters are specified, the name should be left-adjusted within the field and padded on the right with blanks. Alternatively, the name of the phase may be specified as a variable or in an array.

2. CALL OPSYS statement to:

- Modify the block size in the unit assignment table.
- Modify the default BLOCKSIZE (the default is 256).
- Modify the buffer offset in the unit assignment table for ASCII data sets.

Syntax for LANGLVL(66):

Syntax

CALL OPSYS ('FILEOPT',*i*,*j*,*k*,*l*)

Syntax for LANGLVL(77):

Syntax

CALL OPSYS ('VFILOPT',*i*,*j*,*k*,*l*)

FILEOPT
: Required for LANGLVL(66).

VFILOPT
: Required for LANGLVL(77).

i
: Specifies the VS FORTRAN logical unit. A system unit or a unit that has been used for I/O cannot be specified.

j
: Specifies the block length:

  - 18 to 2048 for ASCII
  - 18 to 32767 for EBCDIC

  This length is placed in the UATDBLKS field of the unit assignment table.

k
: Specifies ASCII data sets:

  - A nonzero value indicates an ASCII data set and permits the *l* parameter to be specified.
  - A zero value (or parameter omitted) specifies non-ASCII data sets.

l
: Specifies the buffer offset. This value must only be specified if *k* is a nonzero value. Maximum value is 99. It is placed in UATDRECL of the unit assignment table.

i, *j*, *k*, and *l* may be integer constants, integer variables, or integer array elements.

Because the first parameter is a character string, for LANGLVL(77) the extra parameter that is the address of the length of the first parameter is passed to this module as the second parameter.

Error checking is done to ensure that all values indicated are within the proper limits defined in the FORTRAN ASCII specifications.

Invocation of FILEOPT sets on bit 1, byte 0 of the unit block.

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 VS FORTRAN source. Data is dumped on the error message unit. Variables are dumped automatically upon abnormal termination, or are dumped by program request, on a program unit basis, using CALL SDUMP.

Items displayed are:

- All referenced, local, named variables in their VS FORTRAN-defined data representation

- All variables contained in a blank common, named common, or a dynamic common area in their FORTRAN-defined data representation

- Nonzero or nonblank character array elements only

- Array elements with their correct indexes

The display of data can be invoked both automatically and by program request.

- In the event a task abends (abnormally terminates) in a VS FORTRAN program unit compiled without the NOSDUMP option or with the TEST option, all data in that program unit is automatically dumped.

  Additionally, all data in any VS FORTRAN program unit in the save area traceback chain compiled without the NOSDUMP option or with the TEST option is also dumped. Data occurring in common is dumped at each occurrence, because the data definition in each program unit may be different.

  The display of data follows the IFY240I message and the call chain traceback messages on the object time error unit. The abend SDUMP is done by a copy of routine SDUMP located in module IFYVPOST, which handles post-abend processing.

- Program-requested dumping of data is performed by calling the SDUMP utility program from any program unit. Module IFYSDUMP containing entry SDUMP is loaded in this usage with the calling program.

Syntax

CALL SDUMP [(*rtn*1 [,*rtn*2]...)]

rtn1,rtn2
: The names of other VS FORTRAN program units from which data will be dumped.

Default: Data is dumped only for the calling program when *no* operands are specified.

Note: When using SDUMP to dump variables for *other* routines, those other routine names must be identified on an EXTERNAL statement.

Programming Considerations for SDUMP

- Compilation must be done either without the NOSDUMP option or with the TEST option in order to gain symbolic dump information and location of error information.

- SDUMP for routines not entered has unpredictable results.

- SDUMP for the routine in which the CALL statement is located is done without parameters:

  ```
  CALL SDUMP
  ```

- An EXTERNAL statement must be used to identify the names being passed to SDUMP as external routine names and not local variables.

- The user must not have a routine with the name SDUMP.

- A run-time library containing IFYVPOST must be accessible for post-abend processing.

- At higher levels of optimization (1-3), some variables might not have their true value because of compiler optimization techniques.

- Values for uninitialized variables are unpredictable. The "pass-by-name" subprogram argument(s) in uncalled routines or in subprograms with argument lists shorter than the maximum may cause the SDUMP subroutine to fail.

Examples follow of calling SDUMP from the main program and from a subprogram.

In the main program, the statement

```
EXTERNAL PGM1,PGM2,PGM3
```

would make the address of routines PGM1, PGM2, and PGM3 available for a call to SDUMP:

```
CALL SDUMP (PGM1,PGM2, PGM3)
```

that would cause variables in PGM1, PGM2, and PGM3 to be printed.

In PGM1, the statement

```
EXTERNAL PGM2,PGM3
```

makes PGM2 and PGM3 available. (PGM1 is missing because the call is in PGM1.)

The statements

```
CALL SDUMP
CALL SDUMP (PGM2,PGM3)
```

will dump variables for PGM1, PGM2, and PGM3.

See Appendix H, "Sample Storage Printouts" on page 445, for information about output from symbolic dumps.

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

An integer expression that may have the following values 0 or 1. 0 suppresses program interrupts due to exponent underflow. 1 allows program interrupts due to exponent underflow. The interrupt causes message IFY208I to be produced.

Chapter 10. Extended Error Handling Subroutines and Error Option Table

Extended Error Handling

VS FORTRAN provides five subroutines 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 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 IFYUOPT) 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 VSFUOPT. This macro and information on its use are part of the VS FORTRAN Optional Restricted Materials. Your system administrator will know whether and how the error option table has been modified for your organization.

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 330. For an explanation of how to use extended error handling, see *VS FORTRAN Programming Guide.*

Error Handling Subroutines

ERRMON Subroutine

The ERRMON subroutine calls the FORTRAN error monitor routine, the same routine used by FORTRAN itself when it detects an error.

Syntax

CALL ERRMON (*imes,iretcd,ierno* [,*data*1[,*data*2]...])

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 item 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. If the message length is greater than the length of the buffer, it is printed on two or more lines of printed output.

iretcd

An integer variable made available to the error monitor for setting the following 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: *data*1,*data*2....

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. Because the content of the variables can be altered, the locations in which they are placed should be only in the CALL statement to the error monitor;
otherwise, the user of the function may have literals or variables destroyed.

Because *data*1 and *data*2 are the parameters that the error monitor passes to a user-written routine to correct the detected error, care must be taken to

make sure that these parameters agree in type and number in a call to ERRMON and/or in a call to a user-written corrective routine, if one exists.

ERRMON examines the option table for the appropriate error number and its associated entry and takes the actions specified. If a user-exit address has been specified, ERRMON transfers control to the user-written routine indicated by that address. Thus, the user has the option of handling errors in one of two ways:

- Call ERRMON without supplying a user-written exit routine.
- Call ERRMON and providing a user-written exit routine.

Example:

```
CALL ERRMON (MYMSG,ICODE,315,D1,D2)
```

The example states that the message to be printed is contained in an array named MYMSG; the field named ICODE is to contain the return code; the error condition number to be investigated is 315; and arguments to be passed to the user-written routine are contained in fields named D1 and D2.

ERRSAV Subroutine

The ERRSAV subroutine copies an option table entry into an 8-byte storage area accessible to the VS FORTRAN programmer.

Syntax

CALL ERRSAV (*ierno,tabent*)

ierno
The error number in the option table. Should any number not within the range of the option table be used, an error message will be printed.

tabent
The name of an 8-byte storage area in which the option table entry is to be stored.

Example:

```
CALL ERRSAV (213,ALTERX)
```

The example states that the entry for error number 213 is to be stored in the area named ALTERX.

ERRSET Subroutine

The ERRSET subroutine permits the user to control execution when error conditions occur. For a range of error messages the user can specify:

- How many errors can occur before execution ends
- How many error messages can be printed

- Whether a traceback is to be printed
- Whether a user exit routine is to be executed

Syntax

CALL ERRSET (*ierno,inoa1*[,*inomes*[,*itrace*[,*iusadr*[,*irange*]]]])

ierno

The error number. Should any number not within the range of the option table be used, an error message will be printed. (If *ierno* is specified as 212, there is a special relationship between the *ierno* and *irange* parameters. See the explanation of *irange*, following.)

inoal

An integer specifying the number of errors permitted before each execution is terminated. If *inoal* is specified as either zero or a negative number, the specification is ignored, and the number-of-errors option is not altered. If a value of more than 255 is specified, an unlimited number of errors is permitted.

The value of *inoal* should be set at 2 or greater if transfer of control to a user-supplied error routine is desired after an error. If this parameter is specified with a value of 1, execution is terminated after only one error.

inomes

An integer indicating the number of messages to be printed. A negative value specified for *inomes* suppresses all messages; a specification of zero indicates that the number-of-messages option is not to be altered. If a value greater than 255 is specified, an unlimited number of error messages is permitted.

itrace

An integer whose value may be 0, 1, or 2. A specification of 0 indicates the option is not to be changed; a specification of 1 requests that no traceback be printed after an error; a specification of 2 requests a traceback be printed after each error occurrence. (If a value other 1 or 2 is specified, the option remains unchanged.)

iusadr

Specifies one of the following:

- The value 1, indicating that the option table is to be set to show no user-exit routine (that is, standard corrective action is to be used when continuing execution).
- The name of a closed subroutine that is to be executed after the occurrence of the error identified by *ierno*. The name must appear in an EXTERNAL statement in the source program, and the routine to which control is to be passed must be available at link-editing time.

- The value 0, indicating that the table entry is not to be altered.

See "Coding the User Exit Routine" on page 328.

irange
Specifies one of the following:

- 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, unless *ierno* specifies the number as 212.)

- A print control specifier if *ierno* specified 212. The value 1 is specified to provide single spacing for an overflow line. If a value other than 1 is specified, no print control is provided.

The default value 0 is assumed if the parameter is omitted (that is, no print control is provided, and the values specified for all parameters apply only to the error condition number in *ierno*).

Example 1:

```
CALL ERRSET (310,20,5,0,MYERR,320)
```

This example specifies the following:

- Error condition 310 (*ierno*).
- The error condition may occur up to 20 times (*inoal*).
- The corresponding error message may be printed up to 5 times (*inomes*).
- The current action for traceback information is to remain in force (*itrace*).
- The user-written routine MYERR is to be executed after each error (*iusadr*).
- The same options are to apply to all error conditions from 310 to 320 (*irange*).

Example 2:

```
CALL ERRSET (212,10,5,2,1,1)
```

This example specifies:

- Error condition 212.
- The condition may occur up to 10 times.
- The corresponding message may be displayed up to 5 times.
- Traceback information is to be displayed after each error.
- Standard corrective action is to be executed after an error.
- Print control is to be employed.

For illustration purposes, this example explicitly specifies all default options except that used in requesting print control.

Example 3:

```
CALL ERRSET (212,0,0,0,0,1)
```

This example illustrates an alternative method of specifying exactly the same options as the second example. It states that no changes are to be made to default settings except that used in requesting print control.

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 into 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 40 on page 336, Figure 41 on page 339, and Figure 42 on page 342.

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, DEBUG, PAUSE, or any calls to PDUMP or ERRTRA. Subroutine *x* must not call the library routine that detected the error, or any routine which 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 FORTRAN 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 40 on page 336, Figure 41 on page 339, and Figure 42 on page 342.

3. Only the values 0 and 1 are valid for *iretcd*. A user exit routine must ensure that one of these values is used if it changes the return code setting.

The user-written exit routine can be written in FORTRAN or in assembler language. In either case, it must be able to accept the call to it as shown above. The user exit routine must be a closed subroutine that returns control to the caller.

If the user-written exit routine is written in assembler language, the end of the parameter list can be checked. The high-order bit of the last parameter will be set to 1. Standard register linkage conventions are followed, using registers 13, 14, 15, and 1.

If the routine is written in FORTRAN, the parameter list must match in length the parameter list passed in the CALL statement issued to the error monitor.

Actions that users may take if they wish to correct an error are described in Figure 40 on page 336, Figure 41 on page 339, and Figure 42 on page 342.

ERRSTR Subroutine

The ERRSTR subroutine stores an entry in the option table.

Syntax

CALL ERRSTR (*ierno,tabent*)

ierno
The error number for which the entry is to be stored in the option table. Should any number not within the range of the option table be used, an error will be printed.

tabent
The name of an 8-byte storage area containing the table entry data.

Example:

```
CALL ERRSTR (213,ALTREX)
```

The example states that an entry for error number 213, stored in ALTREX, is to be restored to the option table.

ERRTRA Subroutine

The ERRTRA subroutine dynamically requests a traceback and continued execution.

Syntax

CALL ERRTRA

The CALL ERRTRA statement has no parameters.

Error Option Table

The structure of option table entries is shown in Figure 36 and Figure 37 on page 331. Figure 38 on page 333 lists the preset information for each error condition. Figure 39 on page 334, Figure 40 on page 336, Figure 41 on page 339, and Figure 42 on page 342 summarize the preset information for standard or user-supplied corrective action. The preset entries that cannot be altered are identified in Figure 38 on page 333.

Field Contents	Field Length	Default Value	Field Description
Number of entries	4 bytes	182	Number of entries in the option table.
First Message Number	4 bytes	120	Message number of the first table entry.

Figure 36. Option Table Preface Entry

<table>
<tr><th>FIELD CONTENTS</th><th>FIELD LENGTH</th><th>DEFAULT[1] VALUE</th><th>FIELD DESCRIPTION</th></tr>
<tr><td>Number of error occur-rences</td><td>1 byte</td><td>10[2]</td><td>Number of times this error condition should be allowed to occur. When the value of the error count field (below) equals this value, job processing is terminated. Number may range from 0 to to 255. A value of 0 means an unlimited number of occurrences.[3]</td></tr>
<tr><td>Number messages to print</td><td>1 byte</td><td>5[4]</td><td>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.</td></tr>
<tr><td>Error count</td><td>1 byte</td><td>0</td><td>The number of times this error has occurred. A value of 0 indicates that no occurrences have been encountered.</td></tr>
<tr><td>Option bits</td><td>1 byte</td><td>42 (hex)</td><td>Eight option bits defined as follows (the default setting is underscored):
<table>
<tr><th>Bit</th><th>Setting</th><th>Explanation</th></tr>
<tr><td>0</td><td><u>0</u>
1</td><td>No control character supplied for overflow lines.
Control character supplied to provide single spacing for overflow lines.</td></tr>
<tr><td>1</td><td>0
<u>1</u></td><td>Table entry cannot be modified.[5]
Table entry can be modified.</td></tr>
<tr><td>2</td><td><u>0</u>
1</td><td>Fewer than 256 errors have occurred.
More than 256 errors have occurred. (Add 256 to error count field above to determine the number.)</td></tr>
<tr><td>3[6]</td><td><u>0</u>
1</td><td>Do not print buffer with error message.
Print buffer contents.</td></tr>
<tr><td>4</td><td><u>0</u></td><td>Reserved.</td></tr>
<tr><td>5</td><td><u>0</u>
1</td><td>Print messages default number of times only.
Unlimited printing requested; print for every occurrence of error.</td></tr>
<tr><td>6[7]</td><td>0
<u>1</u></td><td>Do not print traceback map.
Print traceback map.</td></tr>
<tr><td>7</td><td><u>0</u></td><td>Reserved.</td></tr>
</table></td></tr>
<tr><td>User exit</td><td>4 bytes</td><td>1</td><td>Indicates where a user corrective routine is available. A value other than 1 specifies the address of the user-written routine.</td></tr>
</table>

Figure 37. Error Option Table Entry

Notes to Figure 37:

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 may 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 162, 163, 164, 165, 167, 168, 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, 229, and 238.

7 The entry is set to 1 except for error 205.

ERROR CODE	NO. OF ERRORS ALLOWED	NO. OF MESSAGES ALLOWED	PRINT CONTROL	MODI-FIABLE ENTRY	PRINT BUFFER CONTENT	TRACE-BACK ALLOWED	STANDARD CORRECTIVE ACTION
120-139	1	1	NA	Yes	No	Yes	Yes
140	10	5	NA	Yes	No	Yes	Yes
141-145	1	1	NA	No	No	No	No
146-149	1	1	NA	No	No	Yes	No
150	NA	1	NA	Yes	No	No	NA
151	Unlimited	Unlimit	NA	Yes	No	No	NA
152	10	5	NA	Yes	No	Yes	Yes
153	1	1	NA	No	No	Yes	No
154-155	10	5	NA	Yes	No	Yes	Yes
156-158	1	1	NA	No	No	Yes	Yes
159-161	10	5	NA	Yes	No	Yes	Yes
162-165	1	1	NA	No	No	Yes	Yes
166	10	5	NA	Yes	No	Yes	Yes
167	1	1	NA	No	No	Yes	Yes
168	1	1	NA	No	No	Yes	Yes
169-200	10	5	NA	Yes	No	Yes	Yes
201	Unlimited	5	NA	Yes	No	Yes	Yes
202	1	1	NA	No	No	Yes	No
203-204	10	5	NA	Yes	No	Yes	Yes
205	1	1	NA	No	No	No	No
206	10	5	NA	Yes	No	Yes	Yes
207	Unlimited	5	NA	Yes	No	Yes	Yes
208	Unlimited	5	NA	Yes	No	Yes	Yes
209	Unlimited	5	NA	Yes	No	Yes	Yes[1]
210	10	5	NA	Yes	No	Yes	Yes[1]
211	10	5	NA	Yes	No	Yes	Yes
212	10	5	No[2]	Yes	Yes	Yes	Yes
213	10	5	NA	Yes	No	Yes	Yes
214	10	5	NA	Yes	No	Yes	Yes
215	Unlimited	5	NA	Yes	Yes	Yes	Yes
216	10	5	NA	Yes	No	Yes	Yes
217	1[3]	1	NA	Yes	No	Yes	Yes
218	10	4	NA	Yes	Yes[4]	Yes	Yes
219	10[5]	5	NA	Yes	No	Yes	Yes
220	10	5	NA	Yes	No	Yes	Yes
221-225	10	5	NA	Yes	Yes	Yes	Yes
226	10	5	NA	Yes	No	Yes	Yes
227	10	5	NA	Yes	Yes	Yes	Yes
228	10	5	NA	Yes	No	Yes	Yes
229	10	5	NA	Yes	Yes	Yes	Yes
230	1	1	NA	Yes	No	Yes	No
231-237	10	5	NA	Yes	No	Yes	Yes
238	10	5	NA	Yes	Yes	Yes	Yes
239	10	5	NA	Yes	No	Yes	Yes
240	1	1	NA	No	No	Yes	No
241-301	10	5	NA	Yes	No	Yes	Yes

Figure 38. Option Table Default Values

Notes to Figure 38:

[1] No corrective action is taken except to continue execution. For boundary alignment, the corrective action is part of the support for misalignment. For error 209, no user corrective action can be specified.

[2] If a print control character is not supplied, the overflow line is not shifted to incorporate the print control character. Thus, if the device is tape, the data is intact, but if the device is a printer, the first character of the overflow line is not printed but is treated instead as the print control. Unless the user has planned the overflow, the first character would be random and thus the overflow print line control can be any of the possible ones. It is suggested that when the device is a printer, the option be changed to provide single spacing.

[3] It 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.

[4] For an input/output error, the buffer may 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.

[5] Since 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 39

For user-supplied corrective action, see Figure 40 on page 336.

Error Code	IOSTAT=	ERR=
120-129	Yes	Yes
130	Yes[5]	Yes[6]
131-140	Yes	Yes
151	No	No
152	Yes	Yes
153-154	No	No
155	Yes	Yes
156-161	No	No
162-165	Yes	Yes
166	No	No
167	No[5]	No
168	No	No

Figure 39 (Part 1 of 2). IOSTAT and ERR Parameters Honored for I/O Errors

Error Code	IOSTAT=	ERR=
169	No	No
170	Yes	Yes
171	No	No
172-174	Yes	Yes
175-199	No	No
200	Yes	Yes
201	No	No
203-204	No	No
205	No	No
206-213	No	No
214	Yes	Yes
215	No	No
216	Yes	Yes
217	No	No
218-220	Yes	Yes
221-230	No	No
231-232	Yes	Yes
233	No	No
234-236	Yes	Yes
237-285	No	No
286	Yes	Yes
287-301	No	No

Figure 39 (Part 2 of 2). IOSTAT and ERR Parameters Honored for I/O Errors

Notes to Figure 39:

1 If IOSTAT=Yes for a given error number, the IOSTAT variable will be set to the error number if that error occurs.

2 If IOSTAT=No for a given error number, the IOSTAT variable will not be set if the error occurs.

3 If ERR=Yes for a given error number, the ERR label will be branched to if the error occurs.

4 If ERR=No for a given error number, the ERR label will not be branched to if the error occurs.

5 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.

6 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.

Message Corrective Action Cross-Reference Tables

Error Code	Parameters Passed To User Exit	Standard Corrective Action	User-Supplied Corrective Action
120-123	A,B,D	I/O statement ignored.	See Note 1.
124	A,B,D,L	I/O statement ignored.	See Note 1.
125	A,B,D,J,K,L	I/O statement ignored.	See 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.	See Note 1.
129	A,B,D,M,N	The record is not written or rewritten. File position is lost.	See Note 1.
130	A,B,D,O,P	The I/O statement is not processed further. File position is lost.	See Note 1.
131-134	A,B,D	The file was not successfully opened for use by the FORTRAN program even though the VSAM KSDS was internally opened for access via VSAM.	See 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.	See Note 1.

Figure 40 (Part 1 of 3). Corrective Action after Error

Error Code	Parameters Passed To User Exit	Standard Corrective Action	User-Supplied Corrective Action
138	A,B,D	The file was not successfully opened for use by the FORTRAN program even though the VSAM KSDS was internally opened for access via VSAM.	See Note 1.
139	A,B,D,M, N,Q	The record is not rewritten. File position is lost.	See 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.	See Note 1.
205	A,B,D	Program termination.	See Note 1.
206	A,B,I	I=low order part of number for input too large.	User may alter I. See 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	*Input*: Ignore remainder of I/O list. *Output*: Continue by starting new output record. Supply carriage control character if required by Option Table.	See Note 1.
213	A,B,D	Ignore remainder of I/O list.	See Note 1.
214	A,B,D	*Input*: Ignore remainder of I/O list. Ignore input/output request if for ASCII tape. *Output*: If unformatted write initially requested, change record format to VS. If formatted write initially requested, ignore input/output request.	If user correction is requested, the remainder of the I/O list is ignored.
215	A,B,E	Substitute zero for the invalid character.	The character placed in E will be substituted for the invalid character; input/output operations may not be performed. See Note 3.
217	A,B,D	Increase VS FORTRAN sequence number and read next file.	See Note 1.
218	A,B,D,F	Ignore remainder of I/O list.	See Note 1.

Figure 40 (Part 2 of 3). Corrective Action after Error

Error Code	Parameters Passed To User Exit	Standard Corrective Action	User-Supplied Corrective Action
219-224	A,B,D	Ignore remainder of I/O list.	See Note 1.
225	A,B,E	Substitute zero for the invalid character.	The character placed in E will be substituted for the invalid character. See Note 3.
226	A,B,R	R=0 for input number too small. R = 16**63 - 1 for input number too large.	User may alter R.
227	A,B,D	Ignore remainder of I/O list.	See Note 1.
229	A,B,D	Ignore remainder of I/O list.	See Note 1.
231	A,B,D	Ignore remainder of I/O list.	See Note 1.
232	A,B,D,G	Ignore remainder of I/O list.	See Note 1.
233	A,B,D	Change record number to list maximum allowed (32000).	See Note 1.
234, 236	A,B,D	Ignore remainder of I/O list.	See Note 1.
237	A,B,D,F	Ignore remainder of I/O list.	See Note 1.
238	A,B,D	Ignore remainder of I/O list.	See Note 1.
240	See Note 4	Program termination	None
286	A,B,D	Ignore request	See Note 1.
287	A,B,D	Ignore request	See Note 1.
288	A,B,D	Implied wait	See Note 1.

Figure 40 (Part 3 of 3). Corrective Action after Error

Notes to Figure 40:

The alphabetic characters used in the "Parameters Passed to User" column have the following meanings:

Parameter	Meaning
A	Address of return code field (INTEGER*4)
B	Address of error number (INTEGER*4)
C	Address of invalid format character (See Note 5)
D	Address of data set reference number (INTEGER*4)
E	Address of invalid character (See Note 5)
F	Address of DECB
G	Address of record number requested (INTEGER*4)
I	Result after conversion (INTEGER*4)
J	Address of value of key argument
K	Address of length of key argument supplied
L	Address of KEYID value
M	Address of beginning of record
N	Address of length of record

O	Address of VSAM return code
P	Address of error or feedback code
Q	Address of key in record previously read
R	Result after conversion (REAL*4)

Note 1: If the error was not caused during asynchronous input/output processing, the user exit-routine cannot perform any asynchronous I/O operation and, in addition, may not perform any REWIND, BACKSPACE, or ENDFILE operation. If the error was caused during asynchronous input/output processing, the user cannot perform any input/output operation. On return to the library, the remainder of the input/output request will be ignored.

If error condition 218 (input/output error detected) occurs while error messages are being written to the object error data set, the message is written to the console and the job is terminated. If no DD card has been supplied for the object error data set, error message IFY219I is written out at the console and the job is terminated.

Note 2: The user exit routine may supply an alternative answer for the setting of the result register. The routine should always set an INTEGER*4 variable and the FORTRAN library will load fullword or halfword depending on the length of the argument causing the error.

Note 3: Alternatively, the user can set the return code to 0, thus requesting a standard corrective action.

Note 4: Code 240 generates a message showing the system or program code causing program termination, the address of the STAE Control Block, and the contents of the last PSW when abnormal termination occurred. Further information appears under message code IFY240 in Appendix I, "Library Procedures and Messages" on page 463.

Note 5:

- If LANGLVL(66), then LOGICAL*1.
- If LANGLVL(77), then CHARACTER*1.

Error Code	Parameters Passed to User Exit[1]	Reason for Interrupt[2]	Standard Corrective Action	User-Supplied Corrective Action
207	A,B,D,I	Overflow Integer overflow (Interrupt code 8) Exponent overflow (Interrupt code C)[4]	For exponent overflow, the result register is set to the largest floating-point number. The sign of the result register is not altered. No standard fixup for other interrupts.	For exponent overflow, the user may alter D[3].
208	A,B,D,I	Underflow Exponent underflow (Interrupt code D).	The result register is set to zero.	The user may alter D[3].
209	A,B,D,I	Divide check Integer divide (Interrupt code 9) Decimal divide (Interrupt code B) Floating-point divide (Interrupt code F)[4]	For floating-point divide, where N/0 and N=0, the result register is set to 0. Where N ¬ 0, the result register is set to the largest possible floating point number. No standard fixup for other interrupts.	For floating-point divide, the user may alter D[3].
210	A,B	Operation exception (Interrupt code 1) Specification exception (Interrupt code 6) Data exception (Interrupt code 7)	No special corrective action other than correcting boundary misalignment for some specification exceptions.	(See Note[5])

Figure 41. Corrective Action after Program Interrupt

Notes to Figure 41:

1 The variable types and meanings are as follows:

Variable	Type	Meaning[6]
A	INTEGER*4	The return code field.

Variable	Type	Meaning[6]
B	INTEGER*4	The error number.
D	REAL*16	The result register after the interrupt.
I	INTEGER*4	The exponent is an integer value for the number in D. The value in I is not the true exponent, but what was left in the exponent field of the floating-point number after the interrupt.

[2] Program interrupts are described in the appropriate Principles of Operations publication, listed in the preface.

[3] The user exit routine may supply an alternate answer for the setting of the result register. This is accomplished by replacing the value in D. Although the interrupt may be caused by a short, long, or extended floating-point operation, the user exit routine need not be concerned with this. The user exit routine should always set the correct length, but may set a REAL*16 variable and the VS FORTRAN 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 VS FORTRAN program is going to use them, the SUBROUTINE statement may 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,I.

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options: Standard Corrective Action (See Notes 2 and 3)	Options: Parameters Passed to User Exit (See Note 4)
241	K=I**J	I = 0, J ⩽ 0	K=0	A, B, I, J
242	Y=X**I	X=0, I ⩽ 0	If I=0, Y=1 If I < 0, Y=•	A, B, X, I
243	DA=D**I	D=0, I ⩽ 0	If I=0, Y=1 If I < 0, Y=•	A, B, D, I
244	XA=X**Y	X=0, Y ⩽ 0	XA=0	A, B, X, Y
245	DA=D**DB	D=0, DB ⩽ 0	DA=0	A, B, D, DB
246	CA=C**I	C=0+0i, I ⩽ 0	If I=0, C=1+0i If I < 0, C=• +0i	A, B, C, I
247	CDA=CD**I	C=0+0i, I ⩽ 0	If I=0, C=1+0i If I < 0, C=• +0i	A, B, CD, I
248	Q=QA**J	QA=0, J ⩽ 0	J < 0, Q=• J=0, Q=1	A, B, QA, J
249	Q=QA**QB	QA=0, QB ⩽ 0	QB < 0, Q=• QB=0, Q=1	A, B, QA, QB
		QA < 0, QB≠.0	Q=|QA|**QB	
250	Q=QA**QB	$\log_2$(QA)*QB ⩾ 252	Q=•	A, B, QA, QB
251	Y=SQRT (X)	X < 0	Y=$\|X\|^{1/2}$	A, B, X
252	Y=EXP (X)	X > 174.673	Y•	A, B, X
253	Y=ALOG (X)	X=0 X < 0	Y=–• Y=log|X|	A, B, X A, B, X
	Y=ALOG10 (X)	X=0 X < 0	Y=–• Y=$\log_{10}$|X|	A, B, X A, B, X
254	Y=COS (X) Y=SIN (X)	|X| ⩾ $(2^{18})*\pi$	Y=$\sqrt{2}/2$	
255	Y=ATAN2 (X,XA)	X=0. XA=0	Y=0	A, B, X, XA
256	Y=SINH (X) Y=COSH (X)	|H| ⩾ 175.366	Y=(SIGN of X)• Y=•	A, B, X
257	Y=ASIN (X)	|X| > 1	If X>1.0, ASIN(X)=$\pi/2$ If X<–1.0, ASIN(X)=$-\pi/2$	
	Y=ACOS (X)		If X>1.0, ACOS=0 If X<–1.0, ACOS=π	
258	Y=TAN (X) Y=COTAN (X)	|X| ⩾ $(2^{18})*\pi$	Y=1	
259	Y=TAN (X)	X is too close to an odd multiple of $\pi/2$	Y=•	A, B, X
	Y=COTAN (X)	X is too close to a multiple of π	Y=•	

Figure 42 (Part 1 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options	
			Standard Corrective Action (See Notes 2 and 3)	Parameters Passed to User Exit (See Note 4)
260	Q=2**QA	QA > 252	Q=•	A, B, QA
261	DA=DSQRT (D)	D < 0	DA=$\|D\|^{1/2}$	A, B, D
262	DA=DEXP (D)	D > 174.673	D=•	A, B, D
263	DA=DLOG (D)	D=0 D < 0	DA=–• DA=log|X|	A, B, D
	DA=DLOG10 (D)	D=0 D < 0	DA=–• DA=$\log_{10}\|X\|$	A, B, D
264	DA=DSIN (D) DA=DCOS (D)	$\|D\| \geq (2^{50})*\pi$	DA=$\sqrt{2}/2$	A, B, D
265	DA=DATAN2 (D,DB)	D=0, DB=0	DA=0	A, B, D, DB
266	DA=DSINH (D) DA=DCOSH (D)	$\|D\| \geq 175.366$	DA=(SIGN of X)• DA=•	A, B, D
267	DA=DASIN (D)	|D| > 1	If D > 1.0, DASIN=$\pi/2$ If D < –1.0, DASIN=$-\pi/2$	
	DA=DACOS (D)		If D > 1.0, DACOS (D)=0 If D < –1.0, DACOS (D)=π	
268	DA=DTAN (D) DA=DCOTAN (D)	$\|X\| \geq (2^{50})*\pi$	DA=1	A, B, D
269	DA=DTAN (D)	D is too close to an odd multiple of $\pi/2$	DA=•	A, B, D
	DA=DCOTAN (D)	D is too close to a multiple of π	DA=•	A, B, D
For error 270, CQA=X_1 + iX_2				
270	CQ=CQA**J	CQA=0+0i J ≤ 0	J=0, CQ=1+0.i J < 0, CQ=• +0.i	A, B, CQA, J
For errors 271 through 275, C=X_1 + iX_2				
271	Z=CEXP (C)	X_1 > 174.673	Z=•(COS X_2 + iSIN X_2)	A, B, C
272	Z=CEXP (C)	$\|X_2\| \geq (2^{18})*\pi$	Z=e^{X_1} + 0*i	A, B, C
273	Z=CLOG (C)	C=0+0i	Z=–• +0i	A, B, C
274	Z=CSIN (C)	$\|X_1\| \geq (2^{18})*\pi$	Z=0+SINH(X_2)*i	A, B, C
	Z=CCOS (C)		Z=COSH(X_2)+0*i	A, B, C
275	Z=SCIN (C)	X_2 > 174.673	Z=$\frac{•}{2}$(SIN X_1 + iCOS X_1)	A, B, C
	Z=CCOS (C)		Z=• (COS X_1 – iSIN X_1)	

Figure 42 (Part 2 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options: Standard Corrective Action (See Notes 2 and 3)	Options: Parameters Passed to User Exit (See Note 4)
275	Z=CSIN (C)	$X_2 < -174.673$	$Z = \frac{\bullet}{2}(\text{SIN } X_1 - i\text{COS } X_1)$	A, B, C
	Z=CCOS (C)		$Z = \frac{\bullet}{2}(\text{COS } X_1 + i\text{SIN } X_1)$	A, B, C
For errors 276 through 280, $CQ = X_1 + iX_2$				
276	Z=CQEXP (CQ)	$X_1 > 174.673$	$Z = \bullet(\text{COS } X_2 + i\text{SIN } X_2)$	A, B, CQ
277	Z=CQEXP (CQ)	$\lvert X_2 \rvert > 2^{100}$	$Z = e^{X_1} + 0*i$	A, B, CQ
278	Z=CQLOG (CQ)	$CQ = 0 + 0i$	$Z = -\bullet + 0i$	A, B, CQ
279	Z=CQSIN (CQ) Z=CQCOS (CQ)	$\lvert X_1 \rvert \geq 2^{100}$	$Z = 0 + \text{DSINH }(X_2)*i$ $Z = \text{DCOSH }(X_2) + 0*i$	A, B, CQ
280	Z=CQSIN (CQ)	$X_2 > 174.673$	$Z = \frac{\bullet}{2}(\text{SIN } X_1 + i\text{COS } X_1)$	A, B, CQ
	Z=CQCOS (CQ)		$Z = \frac{\bullet}{2}(\text{COS } X_1 = i\text{SIN } X_1)$	A, B, CQ
	Z=CQSIN (CQ) Z=CQCOS (CQ)	$X_2 < -174.673$	$Z = \frac{\bullet}{2}(\text{SIN } X_1 - i\text{COS } X_1)$ $Z = \frac{\bullet}{2}(\text{COS } X_1 = i\text{SIN } X_1)$	A, B, CQ
For errors 281 through 285, $CD = X_1 + iX_2$				
281	Z=CDEXP (CD)	$X_1 > 174.673$	$Z = \bullet(\text{COS } X_2 + i\text{SIN } X_2)$	A, B, CD
282	Z=CDEXP (CD)	$\lvert X_2 \rvert \geq (2^{50})*\pi$	$Z = e^{X_1} + 0*i$	A, B, CD
283	Z=CDLOG (CD)	$CD = 0 + 0i$	$Z = -\bullet + 0i$	A, B, CD
284	Z=CDSIN (CD) Z=CDCOS (CD)	$\lvert X_1 \rvert \geq (2^{50})*\pi$	$Z = 0 + \text{SINH}(X_2)*i$ $Z = \text{COSH}(X_2) + 0*i$	A, B, CD
285	Z=CDSIN (CD)	$X_2 > 174.673$	$Z = \frac{\bullet}{2}(\text{SIN } X_1 + i\text{COS } X_1)$	A, B, CD
	Z=CDCOS (CD)		$Z = \frac{\bullet}{2}(\text{COS } X_1 - i\text{SIN } X_1)$	A, B, CD
	Z=CDSIN (CD)	$X_2 < -174.673$	$Z = \frac{\bullet}{2}(\text{SIN } X_1 - i\text{COS } X_1)$	A, B, CD
	Z=CDCOS (CD)		$Z = \frac{\bullet}{2}(\text{COS } X_1 + i\text{SIN } X_1)$	A, B, CD
289	QA=QSQRT (Q)	$Q < 0$	$QA = \lvert Q \rvert^{1/2}$	A, B, Q
290	Y=GAMMA (X)	$X \leq 2^{-252}$ or $X \geq 57.5744$	$Y = \bullet$	A, B, X
291	Y=ALGAMA (X)	$X \leq 0$ or $X \geq 4.2937*10^{73}$	$Y = \bullet$	A, B, X

Figure 42 (Part 3 of 4). Corrective Action after Mathematical Subroutine Error

Error Code	FORTRAN Reference (See Note 1)	Invalid Argument Range	Options: Standard Corrective Action (See Notes 2 and 3)	Options: Parameters Passed to User Exit (See Note 4)
292	QA=QEXP (Q)	Q>174.673	QA=•	A, B, Q
293	QA=QLOG (Q)	Q=0	QA=–•	A, B, Q
		Q<0	QA=log|X|	
	QA=QLOG10 (Q)	Q=0	QA=•	A, B, Q
		Q<0	$QA=\log_{10}\|X\|$	A, B, Q
294	QA=QSIN (Q) QA=QCOS (Q)	$\|Q\| \geq 2^{100}$	$QA=\sqrt{2}/2$	A, B, Q
295	QA=QATAN2 (Q,QB)	Q=0, QB=0	QA=0	A, B, Q, QB
296	QA=QSINH (Q)	|Q|≥175.366	QA=• (SIGN Q)	A, B, Q
	QA=QCOSH (Q)		QA=•	
297	QA=QARSIN (Q)	|Q|>1	If Q>1.0, QARSIN=π/2	A, B, Q
			If Q<–1.0, QARSIN=–π/2	A, B, Q
	QA=QARCOS (Q)		If Q>1.0, QARCOS(Q)=0 If Q<–1.0, QARCOS(Q)=π	A, B, Q
298	QA=QTAN (Q) QA=QCOTAN (Q)	$\|Q\| > 2^{100}$	QA=1	A, B, Q
299	QA=QTAN (Q)	Q is too close to an odd multiple of π/2	QA=•	A, B, Q
	QA=QCOTAN (Q)	Q is too close to a multiple of π	QA=•	A, B, Q
300	DA=DGAMMA (D)	$D \leq 2^{-252}$ or D≥57.5774	DA=•	A, B, D
301	DA=DLGAMA (D)	D≤0 or $D \geq 4.2937*10^{73}$	DA=•	

Figure 42 (Part 4 of 4). Corrective Action after Mathematical Subroutine Error

Notes to follow:

Notes to Figure 42:

1 The variable types are as follows:

Variable	Type
A,B	INTEGER*4
I,J,K	INTEGER*4
X,XA,Y	REAL*4
D,DA,DB	REAL*8
C,CA	COMPLEX*8
Q,QA,QB	REAL*16
CQ,CQA	COMPLEX*32
Z,X_1,X_2	Complex variables to be given the length of the function argument when they appear.
CD,CDA	COMPLEX*16

2 The largest number that can be represented in floating point is indicated by the symbol •.

3 The value e= 2.7183 (approximately).

4 The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.

Appendix A. Source Language (FIPS) Flagger

The VS FORTRAN compiler can flag FORTRAN statements that do not conform to the syntax of the Full or Subset ANS FORTRAN 1978 Standard. See the ANS manual for subset language flags.

Items Flagged for Full ANS Language

- FREE option.

  The FIPS option cannot be specified with free-form source. The FIPS flagging is ignored.

- LANGLVL(66) option.

  The FIPS option cannot be specified for the 1966 ANS FORTRAN language. The FIPS flagging is ignored.

Global Items Flagged

- Columns 1 to 5 of a continuation card are not blank.
- The currency symbol ($) is used in a name.
- A noncharacter variable has an actual length specified.
- Explicit type specification statements for REAL*16; explicit type specification statements for COMPLEX*16 and COMPLEX*32.
- *n*H in other than a FORMAT statement.
- Hexadecimal constants used as data initialization.

Statements Flagged

- Statements that do not conform to ANS:
  - Asynchronous READ statement
  - Asynchronous WRITE statement
  - AT statement

- DEBUG statement
- DELETE statement
- DISPLAY statement
- EJECT statement
- END DEBUG statement
- INCLUDE statement
- NAMELIST statement
- READ statement with NAMELIST
- REWRITE statement
- TRACE statement
- WAIT statement
- WRITE statement with NAMELIST

• COMMON statement

  Character and noncharacter data in the same common block.

• DATA statement

  The statement appears before the end of the specification statements.

  Q, Z, or *n*H constant is used.

• EQUIVALENCE statement

  Character and noncharacter data in an EQUIVALENCE relationship.

• FORMAT statement

  The Q or Z format codes are used.

• FUNCTION statement

  A length is specified for a real, logical, integer, or complex function.

• IMPLICIT statement

  A length is specified for a real, logical, integer, or complex range.

  The currency symbol ($) is used as an alphabetic character.

- INTEGER, REAL, COMPLEX, LOGICAL type statements

  Data initialization is specified.

Execution-Time Cautions

The following items are not flagged. However, they are items that are open to misinterpretation and may cause confusion.

- Array declarators in DIMENSION, INTEGER, REAL, COMPLEX, DOUBLE PRECISION, CHARACTER, and COMMON statements.

  The value of the lower dimension can exceed the value of the upper dimension when it is an expression.

- ASSIGN statement

  A variable containing a statement number can be used as containing an integer value with unpredictable results.

- Assigned GOTO statement

  The index variable may not contain a statement number that is specified in the list of statement numbers.

- Assignment statement

  A character assignment can be made with unpredictable results into a string that is also used on the right-hand side of the equal sign.

- COMMON statement

  The same common block can contain character variables corresponding to noncharacter variables across subroutines.

  The length of the same common block may not be the same across subroutines.

  The same common block may be initialized in more than one BLOCK DATA.

- DO statement

  The value of the m3 expression can be zero.

  Transfer into an inactive DO loop will produce unpredictable results.

- ENDFILE statement

  Multifiles can be written.

- FUNCTION, SUBROUTINE, ENTRY statements

  The programs must be available.

The programs can be called recursively with unpredictable results.

The number, type, and length of the actual and dummy arguments may not match.

More than one program and/or common block may have the same name.

- IMPLICIT statement

  The same letter is redefined with different type or length.

- OPEN statement

  The file is repositioned at the beginning.

- READ statement on an internal file

  Read records until the end of an array even if the file is one record.

- READ statement with FORMAT

  Data can be read into the nH field of a FORMAT statement.

- Subscript

  Subscript value may be outside the dimension bounds.

- WRITE statement without format on a direct file.

  Spanned records can be written.

Appendix B. IBM and ANS FORTRAN Features

Either the old FORTRAN (LANGLVL(66)) or the current FORTRAN (LANGLVL(77)) compiler option is provided at the time of compilation. The following groups of features are listed in this appendix:

- New ANS FORTRAN 1977 features

  General features
  New statements
  New features in old statements

- Old IBM extensions *now* in ANS FORTRAN 1977
- IBM extensions *not* in ANS FORTRAN 1977
- LANGLVL(66) features not in VS FORTRAN

New ANS FORTRAN 1977 Features

The following new features of the 1977 American National Standard (ANS) FORTRAN (not supported by the old IBM OS and DOS FORTRAN compilers) are supported in VS FORTRAN.

General Features

- May use asterisk comment indicator in column one.
- Comment before continuation is allowed anywhere in the program unit. Blank card is treated as a comment.
- External unit identifier may be an integer expression.
- Direct-access input/output (syntax different from IBM's).
- Storage-to-storage input/output (Internal File).
- Specified ignoring of input blanks.
- Expressions are allowed in output lists.

- Character data type is allowed.
  - May include character substrings.
  - The collating sequence may be altered.
- Subroutines without RETURN.
  - END in subroutine is the same as RETURN.
- Functions (and their entry points) may exist without arguments.
- Dummy argument may be defined if actual argument is in common.
- Array elements are allowed in statement function definitions.
- Array names without subscripts are allowed in the EQUIVALENCE statement.
- Complex data may be defined through real components.
- Variables used in adjustable dimensions and lengths may be redefined with no effect on size of array.
- Integer expressions are allowed in array declarators.
- Nonunity lower bounds for arrays are allowed.
- Nonpositive subscript values are allowed.
- Named BLOCK DATA subroutines are allowed.
- Executable statements that cannot be reached are allowed.
- ANINT, CHAR, DNINT, DPROD, ICHAR, IDNINT, INDEX, LEN, and NINT are recognized as VS FORTRAN-supplied function names.
- DARCOS and DARSIN functions have different names: DACOS and DASIN.
- Logical operators .EQV. and .NEQV. are allowed.
- A number is permitted on nonexecutable statements.
- Comparison of complex operands with equal and not equal relationals is allowed.
- Exponentiation of complex with complex is allowed.
- All specification statements must precede all DATA statements.
- Negative values for input or output unit identifiers are prohibited.
- Literal format cannot be used for input.

- H format cannot be used for input.
- Use of a slash as a value separator in list-directed input is allowed.
- Character function is allowed.
- Unspecified width is allowed in A format.

New Statements

- Block IF, ELSE IF, ELSE, END IF statements
- Character type statement
- CLOSE statement
- Double precision type statement
- INQUIRE statement
- INTRINSIC statement
- OPEN statement
- PARAMETER statement
- PROGRAM statement
- SAVE statement

New Features in Old Statements

- BACKSPACE statement:
  - UNIT, ERR, and IOSTAT may be used.
- COMMON statement:
  - Commas are optional.
- DATA statement:
  - Implied DO statement is allowed.
  - Type conversion is allowed.
  - Commas after nonterminal slashes are optional.
- DIMENSION statement:
  - Specification can be negative or zero.
  - Both lower and upper bound can be names of constants or expressions.

- DO statement:
  - Loops may be indexed by nonpositive values.
  - Loops may be indexed by integer, real, or double precision values.
  - Index may be decremented by negative values.
  - Backward loops may be used.
  - Zero trip loops may be used.
  - Control variable is defined on exit from loop.
  - Control variable may be real or double precision.
  - Terminal statements are allowed with computed GO TO, PAUSE, or logical IF. They are not allowed with block IF, ELSE IF, ELSE, END IF, or END. If terminal statement is a logical IF, it may not contain a DO, block IF, ELSE IF, ELSE, END IF, or END.
  - Comma is optional following terminal statement number and before control variable.
  - Parameters may be any arithmetic type expression except complex.
  - Parameters may be redefined in loop with no effect on loop control.
  - A block IF statement in the DO range must be entirely within the range of the DO.
  - The range of a DO within a block IF must be entirely contained within the block.
  - DO may be ended by any fall-through statement.
- END statement:
  - May be numbered.
  - Implies STOP or RETURN.
  - Is executable.
- ENDFILE statement:
  - UNIT, ERR, and IOSTAT may be used.
- EXTERNAL statement:

  An ampersand (&) character as the first character of a name is not permitted for compiler option LANGLVL(77). Any name that appears in an EXTERNAL statement is considered as the name of a user-supplied subroutine.

- FORMAT statement:
  - BN and BZ specify ignoring of input blanks.
  - Unlimited parentheses may be used.
  - The label ASSIGNED may be the number of a FORMAT statement.
  - Field width is optional in A*w*.
  - Explicit *n*P scale factor may be used.
  - Ew.dEe, Gw.dEe, Iw.d, SP, SS, S, TLc, and TRc field descriptors may be used.
  - Colon may be used as scan terminator.
  - Optional commas may be used with slashes and colons.
- GO TO statement, assigned:
  - List of statement numbers is optional.
  - Comma outside parentheses is optional.
- GO TO statement, computed:
  - Index may be an integer expression.
  - Comma may be outside parentheses.
- IMPLICIT statement:
  - More than one may be used in a program unit.
  - IMPLICIT may be preceded by ENTRY, FORMAT, or PARAMETER statements and must precede all other specification statements except PARAMETER statements.
  - Double precision and character type statements are included.
- PRINT statement:
  - FORMAT designator may be a character constant.
- READ statements:
  - FORMAT designator may be a character constant.
  - UNIT, ERR, and IOSTAT may be used.
- RETURN statement:
  - Index may be an integer expression.

- REWIND statement:
  - UNIT, ERR, and IOSTAT may be used.
- STOP statement:
  - Quoted literal is allowed.
  - A character constant is permitted.
- Auxiliary input and output statements:
  - UNIT and ERR may be used.
- WRITE statement:
  - May not be used after ENDFILE in sequential input or output.
  - FORMAT designator may be a character constant.
  - UNIT, FMT, REC, and IOSTAT may be used.

Old IBM Extensions Now in ANS FORTRAN 1977

The following items supported as IBM extensions in old IBM OS and DOS FORTRAN compilers *are* now part of the 1977 ANS FORTRAN language. These items are also supported in VS FORTRAN.

- Literals are enclosed in apostrophes.
- STOP and PAUSE statements:
  - Decimal digits are supported.
  - STOP statement string is accessible.
  - Quoted literal in PAUSE statement is supported.
- T format is accepted as a field descriptor.
- Computed GO TO index out of range.
- All combinations of arithmetics across equal sign.
- Mixed-mode arithmetic.
- Mixed-mode relationals.
- Successive exponentiations.
- Generalized subscripts.

- Seven-dimensional arrays.
- END in READ.
- ERR in READ and WRITE.
- Short form of READ and PRINT.
- Sequential list-directed input/output.
- Asterisks for undersized output fields.
- IMPLICIT statement.
- Array names in DATA statement.
- ENTRY statement.
- Alternative returns from subroutines.
- Function and entry names in type statements.
- Generic facility.
- Additional processor-supplied functions.

IBM Extensions Not in ANS FORTRAN 1977

The following IBM extensions are supported by old IBM OS and DOS FORTRAN compilers but are *not* part of the 1977 ANS FORTRAN. They will continue to be supported in VS FORTRAN as IBM extensions.

Some of the following features are available only under the compiler option described in the next section, "LANGLVL(66) Features Not in VS FORTRAN."

- NAMELIST statement.
- Double precision complex.
- Z and Q format descriptor.
- G format for integer and logical.
- ALGAMA, ARCOS, ARSIN, CCOS, CDABS, CDCOS, CDEXP, CDLOG, CDSIN, CDSQRT, COTAN, CQABS, CQCOS, CQEXP, CQLOG, CQSIN, CQSQRT, DARCOS, DARSIN, DBLEQ, DCMPLX, DCONJG, DCOTAN, DERFC, DERF, DFLOAT, DGAMMA, DIMAG, DLGAMA, DREAL, ERF, ERFC, GAMMA, HFIX, IMAG, IQINT, LGAMMA, QABS, QARCOS, QARSIN, QATAN, QATAN2, QCMPLX, QCONJG, QCOSH, QCOS, QCOTAN, QDIM, QERFC, QERF, QEXP, QEXTD, QEXT, QFLOAT, QIMAG, QINT, QLOG, QLOG10, QMAX1, QMIN1, QMOD, QREAL, QSIGN, QSINH, QSIN, QSQRT, QTANH, QTAN, SNGLQ.

- CALL DVCHK, CALL DUMP/PDUMP, CALL EXIT, CALL OVERFL.
- Asynchronous READ, WRITE, and WAIT.
- Extended precision for REAL and COMPLEX.
- Extended debug facility.
- Hexadecimal constants and Z format are allowed.
- Free-form source statements.
- The currency symbol ($) used as alphabetic character.
- Data initialization in type specification statements.
- Optional length specification in specification statements (integer, real, complex, logical) and in FUNCTION statements.
- Mixed-mode expressions involving complex and double precision.
- FORMAT identifier may be an array name (other than character type).
- Continuation line may have anything in columns 1 through 5 other than "C" in column 1.
- RETURN statement is the same as STOP in a main program.
- Partitioned data sets.
- Closing of data set on ABEND.
- STOP*n* is allowed, where *n* equals a return code.
- Initialization with hexadecimal constants.
- EQUIVALENCE statement allows equivalencing of character and noncharacter data types.
- COMMON statement allows character and noncharacter data types in the same common block.

LANGLVL(66) Features Not in LANGLVL(77)

LANGLVL(66) instructs the compiler to compile a program according to the 1966 FORTRAN language. Listed here are some of the features of LANGLVL(66) that are *not* in LANGLVL(77).

- Character constants may 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 only be specified as a compiler option.

- Arguments are received by value.

- Dummy arguments can be enclosed in slashes.

- DARCOS and DARSIN used as function names are recognized as VS FORTRAN-supplied functions; DACOS and DASIN are recognized as user-supplied function names.

- DEFINE FILE statement.

- DO statement and implied DO in I/O. (Loops are always executed at least once.)

- EQUIVALENCE statement. (Accept a multidimensional array with one subscript.)

- EXTERNAL statement: If a VS FORTRAN-supplied function name 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 VS FORTRAN-supplied function name 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.

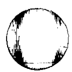
- FIND statement.

- 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 VS FORTRAN-supplied functions will be recognized as generic; if GENERIC is not specified, the automatic function selection facility will not be in effect.

- IBM direct-access READ and WRITE.

- PUNCH *b*, list.



Appendix C. EBCDIC and ISCII/ASCII Codes

The table below contains EBCDIC and ISCII/ASCII codes, where:

EBCDIC refers to IBM EBCDIC code point ordering for the 256 character set.

ISO 8 bit refers to ISO 2022 code point ordering for the 256 character set.

ISCII/ASCII 7 bit refers to ANSI X3.4-1977 code point ordering for the 128 character set.

ISCII/ASCII 6 bit refers to ANSI X3.32-1973 code point ordering for the 64 character set.

The column used for the lexical intrinsic functions is ISCII/ASCII 7 bit.

The blank character to be used to extend character strings for the the intrinsic functions LGE, LGT, LLE, and LLT is the ISCII/ASCII blank (HEX 20).

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ISCII/ ASCII 7 bit for ICHAR	ISCII/ ASCII 6 bit for ICHAR
00	0	NUL	Null	0	0	Note 2
01	1	SOH	Start of heading	1	1	Note 2
02	2	STX	Start of text	2	2	Note 2
03	3	ETX	End of text	3	3	Note 2
04	4	SEL	Select	156	Note 1	Note 2
05	5	HT	Horizontal Tab	9	9	Note 2
06	6	RNL	Required new line	134	Note 1	Note 2
07	7	DEL	Delete	127	127	Note 2
08	8	GE	Graphic Escape	151	Note 1	Note 2
09	9	SPS	Superscript	141	Note 1	Note 2
0A	10	RPT	Repeat	142	Note 1	Note 2
0B	11	VT	Vertical Tab	11	11	Note 2
0C	12	FF	Form Feed	12	12	Note 2
0D	13	CR	Carriage Return	13	13	Note 2
0E	14	SO	Shift out	14	14	Note 2
0F	15	SI	Shift in	15	15	Note 2
10	16	DLE	Data link escape	16	16	Note 2
11	17	DC1	Device control 1	17	17	Note 2
12	18	DC2	Device control 2	18	18	Note 2
13	19	DC3	Device control 3	19	19	Note 2

Notes:

1. This position does not exist in ANSI X3.4-1977 for 7-bit code.

2. This position does not exist in ANSI X3.32-1973 for 6-bit code.

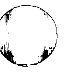
HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ISCII/ ASCII 7 bit for ICHAR	ISCII/ ASCII 6 bit for ICHAR
14	20	RES	Restore	157	Note 1	Note 2
		ENP	Enable presentation			
15	21	NL	New line acknowledgement	133	Note 1	Note 2
16	22	BS	Backspace	8	8	Note 2
17	23	POC	Program-operator communication	135	Note 1	Note 2
18	24	CAN	Cancel	24	24	Note 2
19	25	EM	End of Medium	25	25	Note 2
1A	26	UBS	Unit backspace	146	Note 1	Note 2
1B	27	CU1	Customer use 1	143	Note 1	Note 2
1C	28	IFS	Interchange file separator	28	28	Note 2
1D	29	IGS	Interchange group separator	29	29	Note 2
1E	30	IRS	Interchange record separator	30	30	Note 2
1F	31	IUS	Interchange unit separator	31	31	Note 2
		ITB	Intermediate trans. block			
20	32	DS	Digit select	128	Note 1	Note 2
21	33	SOS	Start of significance	129	Note 1	Note 2
22	34	FS	Field separator	130	Note 1	Note 2
23	35	WUS	Word underscore	131	Note 1	Note 2
24	36	BYP	Bypass	132	Note 1	Note 2
		INP	Inhibit presentation			
25	37	LF	Line feed	10	10	Note 2
26	38	ETB	End of trans. block	23	23	Note 2
27	39	ESC	Escape	27	27	Note 2
28	40		Reserved	136	Note 1	Note 2
29	41		Reserved	137	Note 1	Note 2
2A	42	SM, SW	Set mode, Switch	138	Note 1	Note 2
2B	43	FMT	Format	139	Note 1	Note 2
2C	44		Reserved	140	Note 1	Note 2
2D	45	ENQ	Enquiry	5	5	Note 2
2E	46	ACK	Acknowledge	6	6	Note 2
2F	47	BEL	Bell	7	7	Note 2
30	48		Reserved	144	Note 1	Note 2
31	49		Reserved	145	Note 1	Note 2
32	50	SYN	Synchronous	22	22	Note 2
33	51	IR	Index	147	Note 1	Note 2
34	52	PP	Presentation position	148	Note 1	Note 2
35	53	TRN	Transparent	149	Note 1	Note 2
36	54	NBS	Numeric backspace	150	Note 1	Note 2
37	55	EOT	End of transmission	4	4	Note 2
38	56	SBS	Subscript	152	Note 1	Note 2
39	57	IT	Indent	153	Note 1	Note 2
3A	58	RFF	Required	154	Note 1	Note 2
3B	59	CU3	Customer use 3	155	Note 1	Note 2

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ISCII/ ASCII 7 bit for ICHAR	ISCII/ ASCII 6 bit for ICHAR
3C	60	DC4	Device code 4	20	20	Note 2
3D	61	NAK	Negative acknowledge	21	21	Note 2
3E	62		Reserved	158	Note 1	Note 2
3F	63	SUB	Substitute	26	26	Note 2
40	64	SP	Space	32	32	0
41	65	RSP	Required space	160	Note 1	Note 2
42	66			161	Note 1	Note 2
43	67			162	Note 1	Note 2
44	68			163	Note 1	Note 2
45	69			164	Note 1	Note 2
46	70			165	Note 1	Note 2
47	71			166	Note 1	Note 2
48	72			167	Note 1	Note 2
49	73			168	Note 1	Note 2
4A	74	¢	Cent sign	91	91	59
4B	75	.	Period, decimal point	46	46	14
4C	76	<	Less-than sign	60	60	28
4D	77	(	Left parenthesis	40	40	8
4E	78	+	Plus sign	43	43	11
4F	79	|	Logical OR	33	33	1
50	80	&	Ampersand	38	38	6
51	81			169	Note 1	Note 2
52	82			170	Note 1	Note 2
53	83			171	Note 1	Note 2
54	84			172	Note 1	Note 2
55	85			173	Note 1	Note 2
56	86			174	Note 1	Note 2
57	87			175	Note 1	Note 2
58	88			176	Note 1	Note 2
59	89			177	Note 1	Note 2
5A	90	!	Exclamation point	93	93	61
5B	91	$	Currency symbol	36	36	4
5C	92	*	Asterisk	42	42	10
5D	93	)	Right parenthesis	41	41	9
5E	94	;	Semicolon	59	59	27
5F	95	¬	Logical NOT	94	94	62
60	96	-	Minus sign, Hyphen	45	45	13
61	97	/	Slash	47	47	15
62	98			178	Note 1	Note 2
63	99			179	Note 1	Note 2
64	100			180	Note 1	Note 2
65	101			181	Note 1	Note 2
66	102			182	Note 1	Note 2
67	103			183	Note 1	Note 2
68	104			184	Note 1	Note 2
69	105			185	Note 1	Note 2
6A	106	|	Vertical line	124	124	Note 2
6B	107	,	Comma	44	44	12
6C	108	%	Percent sign	37	37	5
6D	109		Underscore	95	95	63

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ISCII/ ASCII 7 bit for ICHAR	ISCII/ ASCII 6 bit for ICHAR
6E	110	>	Greater-than sign	62	62	30
6F	111	?	Question mark	63	63	31
70	112			186	Note 1	Note 2
71	113			187	Note 1	Note 2
72	114			188	Note 1	Note 2
73	115			189	Note 1	Note 2
74	116			190	Note 1	Note 2
75	117			191	Note 1	Note 2
76	118			192	Note 1	Note 2
77	119			193	Note 1	Note 2
78	120			194	Note 1	Note 2
79	121	GRA	Grave accent	96	96	Note 2
7A	122	:	Colon	58	58	26
7B	123	#	Number sign	35	35	3
7C	124	@	At sign	64	64	32
7D	125	'	Prime, Apostrophe	39	39	7
7E	126	=	Equal sign	61	61	29
7F	127	"	Quotation marks	34	34	2
80	128			195	Note 1	Note 2
81	129	a	Lowercase a	97	97	Note 2
82	130	b	Lowercase b	98	98	Note 2
83	131	c	Lowercase c	99	99	Note 2
84	132	d	Lowercase d	100	100	Note 2
85	133	e	Lowercase e	101	101	Note 2
86	134	f	Lowercase f	102	102	Note 2
87	135	g	Lowercase g	103	103	Note 2
88	136	h	Lowercase h	104	104	Note 2
89	137	i	Lowercase i	105	105	Note 2
8A	138			196	Note 1	Note 2
8B	139			197	Note 1	Note 2
8C	140			198	Note 1	Note 2
8D	141			199	Note 1	Note 2
8E	142			200	Note 1	Note 2
8F	143			201	Note 1	Note 2
90	144			202	Note 1	Note 2
91	145	j	Lowercase j	106	106	Note 2
92	146	k	Lowercase k	107	107	Note 2
93	147	l	Lowercase l	108	108	Note 2
94	148	m	Lowercase m	109	109	Note 2
95	149	n	Lowercase n	110	110	Note 2
96	150	o	Lowercase o	111	111	Note 2
97	151	p	Lowercase p	112	112	Note 2
98	152	q	Lowercase q	113	113	Note 2
99	153	r	Lowercase r	114	114	Note 2
9A	154			203	Note 1	Note 2
9B	155			204	Note 1	Note 2
9C	156			205	Note 1	Note 2
9D	157			206	Note 1	Note 2
9E	158			207	Note 1	Note 2
9F	159			208	Note 1	Note 2

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ISCII/ ASCII 7 bit for ICHAR	ISCII/ ASCII 6 bit for ICHAR
A0	160			209	Note 1	Note 2
A1	161	TIL	Tilde	126	126	Note 2
A2	162	s	Lowercase s	115	115	Note 2
A3	163	t	Lowercase t	116	116	Note 2
A4	164	u	Lowercase u	117	117	Note 2
A5	165	v	Lowercase v	118	118	Note 2
A6	166	w	Lowercase w	119	119	Note 2
A7	167	x	Lowercase x	120	120	Note 2
A8	168	y	Lowercase y	121	121	Note 2
A9	169	z	Lowercase z	122	122	Note 2
AA	170			210	Note 1	Note 2
AB	171			211	Note 1	Note 2
AC	172			212	Note 1	Note 2
AD	173			213	Note 1	Note 2
AE	174			214	Note 1	Note 2
AF	175			215	Note 1	Note 2
B0	176			216	Note 1	Note 2
B1	177			217	Note 1	Note 2
B2	178			218	Note 1	Note 2
B3	179			219	Note 1	Note 2
B4	180			220	Note 1	Note 2
B5	181			221	Note 1	Note 2
B6	182			222	Note 1	Note 2
B7	183			223	Note 1	Note 2
B8	184			224	Note 1	Note 2
B9	185			225	Note 1	Note 2
BA	186			226	Note 1	Note 2
BB	187			227	Note 1	Note 2
BC	188			228	Note 1	Note 2
BD	189			229	Note 1	Note 2
BE	190			230	Note 1	Note 2
BF	191			231	Note 1	Note 2
C0	192	{	Opening brace	123	123	Note 2
C1	193	A	Uppercase A	65	65	33
C2	194	B	Uppercase B	66	66	34
C3	195	C	Uppercase C	67	67	35
C4	196	D	Uppercase D	68	68	36
C5	197	E	Uppercase E	69	69	37
C6	198	F	Uppercase F	70	70	38
C7	199	G	Uppercase G	71	71	39
C8	200	H	Uppercase H	72	72	40
C9	201	I	Uppercase I	73	73	41
CA	202			232	Note 1	Note 2
CB	203			233	Note 1	Note 2
CC	204			234	Note 1	Note 2
CD	205			235	Note 1	Note 2
CE	206			236	Note 1	Note 2
CF	207			237	Note 1	Note 2
D0	208	}	Closing brace	125	125	Note 2
D1	209	J	Uppercase J	74	74	42

HEX Code	Ordinal Position for ICHAR	EBCDIC Graphic or Control	Description	ISO 8 bit for ICHAR	ISCII/ ASCII 7 bit for ICHAR	ISCII/ ASCII 6 bit for ICHAR
D2	210	K	Uppercase K	75	75	43
D3	211	L	Uppercase L	76	76	44
D4	212	M	Uppercase M	77	77	45
D5	213	N	Uppercase N	78	78	46
D6	214	O	Uppercase O	79	79	47
D7	215	P	Uppercase P	80	80	48
D8	216	Q	Uppercase Q	81	81	49
D9	217	R	Uppercase R	82	82	50
DA	218			238	Note 1	Note 2
DB	219			239	Note 1	Note 2
DC	220			240	Note 1	Note 2
DD	221			241	Note 1	Note 2
DE	222			242	Note 1	Note 2
DF	223			243	Note 1	Note 2
E0	224	\	Reverse slash	92	92	60
E1	225			159	Note 1	Note 2
E2	226	S	Uppercase S	83	83	51
E3	227	T	Uppercase T	84	84	52
E4	228	U	Uppercase U	85	85	53
E5	229	V	Uppercase V	86	86	54
E6	230	W	Uppercase W	87	87	55
E7	231	X	Uppercase X	88	88	56
E8	232	Y	Uppercase Y	89	89	57
E9	233	Z	Uppercase Z	90	90	58
EA	234			244	Note 1	Note 2
EB	235			245	Note 1	Note 2
EC	236			246	Note 1	Note 2
ED	237			247	Note 1	Note 2
EE	238			248	Note 1	Note 2
EF	239			249	Note 1	Note 2
F0	240	0	Zero	48	48	16
F1	241	1	One	49	49	17
F2	242	2	Two	50	50	18
F3	243	3	Three	51	51	19
F4	244	4	Four	52	52	20
F5	245	5	Five	53	53	21
F6	246	6	Six	54	54	22
F7	247	7	Seven	55	55	23
F8	248	8	Eight	56	56	24
F9	249	9	Nine	57	57	25
FA	250	|	Long vertical mark	250	Note 1	Note 2
FB	251			251	Note 1	Note 2
FC	252			252	Note 1	Note 2
FD	253			253	Note 1	Note 2
FE	254			254	Note 1	Note 2
FF	255	E0	Eight ones	255	Note 1	Note 2



Appendix D. Algorithms for Library Mathematical Functions

This appendix contains information about the method by which each mathematical function is computed. The information for explicitly called subprograms is arranged alphabetically according to the specific function of each subprogram (that is, absolute value, exponentiation, logarithmic, etc.). The individual entry names associated with each subprogram are arranged logically from simple to complex within each function. For example, the heading "Square Root Subprograms" will have algorithms arranged in the following order by entry name: SQRT, DSQRT, CSQRT, CDSQRT.

Information for the implicitly called subprograms is arranged alphabetically according to function, and alphabetically by entry name within that function. For example, the heading "Complex Multiply and Divide Subprograms" will have algorithms arranged in the following order: CDDVD#/CDMPY#, CDVD#/CMPY#.

The information for each subprogram is divided into two parts. The first part describes the algorithms used; the second part describes the effect of an argument error upon the accuracy of the answer returned.

The presentation of each algorithm is divided into its major computational steps; the formulas necessary for each step are supplied. For the sake of brevity, the needed constants are normally given only symbolically. Some of the formulas are widely known; those that are not so widely known are derived from more common formulas. The process leading from the common formula to the computational formula is sketched in enough detail so that the derivation can be reconstructed by anyone who has an understanding of college mathematics and access to the common texts of numerical analysis. Many approximations were derived by the so-called "minimax" methods. The approximation sought by these methods can be characterized as follows. Given a function $f(x)$, an interval I, the form of the approximation (such as the rational form with specified degrees), and the type of error to be minimized (such as the relative error), there is normally a unique approximation to $f(x)$ whose maximum error over I is the smallest among all possible approximations of the given form. Details of the theory and the various methods of deriving such approximation are provided in standard references. The accuracy figures cited in the algorithm sections are theoretical, and they do not take round-off errors into account. Minor programming techniques used to minimize round-off errors are not described here.

The accuracy of an answer provided by these algorithms is influenced by two factors: the performance of the subprogram (see Appendix F, "Accuracy Statistics" on page 425) and the accuracy of the argument. The effect of an argument error upon the accuracy of an answer depends solely upon the mathematical function involved and not upon the particular coding used in the subprogram.

A guide to the propagational effect of argument errors is provided because argument errors always influence the accuracy of answers whether the errors are accumulated prior to use of the subprogram or introduced by newly converted data. 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 appendix to describe the effect of an argument error upon the accuracy of the answer:

SYMBOL	EXPLANATION	
$g(x)$	The result given by the subprogram.	
$f(x)$	The correct result.	
ϵ	$\left\lvert\frac{f(x) - g(x)}{f(x)}\right\rvert$	The relative error of the result given by the subprogram.
δ	The relative error of the argument.	
E	$\lvert f(x) - g(x)\rvert$	The absolute error of the result given by the subprogram.
Δ	The absolute error of the argument.	

The notation used for the continued fractions complies with the specifications set by the United States National Bureau of Standards.[1]

Although it is not specifically stated below for each subroutine, the algorithms in the appendix were programmed to conform to the following standards governing floating-point overflow/underflow.

- Intermediate underflow 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 subroutine. This rule does not apply to CDABS and CABS.

- When the magnitude of the answer is less than 16 to the minus 65th power, zero 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 mathematical functions have been coded with careful control of error situations. A result is provided whenever the answer is within the range representable in the floating-point form. In order to be consistent with VS FORTRAN control of exponent overflow/underflow exceptions, the following types of conditions are recognized and handled separately.

---

1 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).

When the magnitude of the function value is too large to be represented in the floating-point form, the condition is called a terminal overflow; when the magnitude is too small to be represented, a terminal underflow. On the other hand, if the function value is representable, but if execution of the chosen algorithm causes an overflow or underflow in the process, this condition is called an intermediate overflow or underflow.

Function subroutines in the VS FORTRAN library have been coded to observe the following rules for these conditions:

1. Algorithms which can cause an intermediate overflow have been avoided. Therefore, an intermediate overflow should occur only rarely during the execution of a function subroutine 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 subroutine. 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 of underflow in the same manner as for an arithmetic statement. Terminal underflows can occur in the following function subroutines: EXP, DEXP, ATAN2, DATAN2, ERFC, DERFC, QATAN2, and QEXP.

For implicit arithmetic subroutines, these rules do not apply. In this case, both terminal overflows and terminal underflows will cause respective floating-point exceptions. In addition, in the case of complex arithmetic (implicit multiply and divide), premature overflow/underflow is possible when the result of arithmetic is very close to an overflow or underflow condition.

The algorithms for the alternative mathematical library subroutines can be found in the articles listed in the preface.

Explicitly Called Subprograms

Absolute Value Subprograms

ABS/IABS/DABS/QABS

Algorithm

If $x < 0$, $|x| = -x$. Otherwise $|x| = x$.

CABS/CDABS

Algorithm

1. Write $|x + iy| = a + ib$.
2. Let $v_1 = \max(|x|, |y|)$, and $v_2 = \min(|x|, |y|)$.
3. If characteristics of v_1 and v_2 differ by 7 (15 for CDABS) or more, or if $v_2 = 0$, then $a = v_1$, $b = 0$.
4. Otherwise,

$$a = 2 \cdot v_1 \cdot \sqrt{1/4 + 1/4 \left(\frac{v_2}{v_1}\right)^2}, \text{ and } b = 0.$$

If the answer is greater than 16^{63}, the floating-point overflow interruption will take place (see Appendix C). The algorithms for both complex absolute value subprograms are identical. Each subprogram uses the appropriate real square root subprogram (SQRT or DSQRT).

CQABS

Algorithm

1. Write $|x + iy| = a + ib$.
2. Let $v_1 = \max(|x|, |y|)$, and $v_2 = \min(|x|, |y|)$.
   Let $16^{p-1} \leqq v_1 < 16^p$.
3. If characteristics of v_1 and v_2 differ by 15 or more, or if $v_2 = 0$, then $a = v_1$, $b = 0$.
4. Otherwise, let $w_1 = 16^{1-p} \cdot v_1$, and $w_2 = 16^{1-p} \cdot v_2$.
5. Compute $w = \sqrt{w_1{}^2 + w_2{}^2}$. Then $a = 16^{p-1}\, w$ and $b = 0$.
6. The scaling factor 16^{p-1} is easy to construct. Scaling is carried out by short precision divisions, and the restoration is carried out by extended precision multiplication.

Effect of an Argument Error

$\epsilon \sim \frac{x^2}{a^2}\,\delta(x) + \frac{y^2}{a^2}\,\delta(y)$ 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 Subprograms

ASIN/ACOS

Algorithm

1. If $0 \leqq x \leqq \frac{1}{2}$, then compute arcsin (x) by a continued fraction of the form:

   $$\arcsin(x) \cong x + x^3 \cdot F \text{ where}$$

   $$F = \frac{d_1}{(x^2 + c_1) +} \; \frac{d_2}{(x^2 + c_2)}.$$

   The coefficients of this formula were derived by transforming the minimax rational approximation (in relative error, over the range $0 \leqq x^2 \leqq \frac{1}{4}$) for arcsin $(x)/x$ of the following form:

   $$\frac{\arcsin(x)}{x} \cong a_0 + x^2 \cdot \left[\frac{a_1 + a_2 x^2}{b_0 + b_1 x^2 + x^4}\right].$$

   Minimax was taken under the constraint that $a_0 = 1$ exactly. The relative error of this approximation is less than $2^{-28.3}$.
   If $0 \leqq x \leqq \frac{1}{2}$, arccos$(x)$ is computed as:

   $$\arccos(x) = \frac{\pi}{2} - \arcsin(x).$$

2. If $\frac{1}{2} < x \leqq 1$, then compute arccos(x) essentially as:

   $$\arccos(x) = 2 \cdot \arcsin\left(\sqrt{\frac{1-x}{2}}\right).$$

   This case is now reduced to the first case because within these limits,

   $$0 \leqq \sqrt{\frac{1-x}{2}} \leqq \frac{1}{2}.$$

   This computation uses the real square root subprogram (SQRT).
   If $\frac{1}{2} < x \leqq 1$, arcsin(x) is computed as:

   $$\arcsin(x) = \frac{\pi}{2} - \arccos(x).$$

   Implementation of the above algorithms (steps 1 and 2) was carried out with care to minimize the round-off errors.

3. If $-1 \leqq x < 0$, then $\arcsin(x) = -\arcsin|x|$
   and $\arccos(x) = \pi - \arccos|x|$.

   This reduces these cases to one of the two positive cases.

Effect of an Argument Error

$E \sim \frac{\Delta}{\sqrt{1 - x^2}}$. For small values of x, $E \sim \Delta$. Toward the limits (± 1) of the range, a small Δ causes a substantial error in the answer. For the arcsine, $\epsilon \sim \delta$ if the value of x is small.

Algorithm

1. If $0 \leqq x \leqq 1/2$, then compute $\arcsin(x)$ by a continued fraction of the form:

   $\arcsin(x) \cong x + x^3 \cdot F$ where

   $$F = c_1 + \frac{d_1}{(x^2 + c_2) +} \frac{d_2}{(x^2 + c_3) +} \frac{d_3}{(x^2 + c_4) +} \frac{d_4}{(x^2 + c_5)}.$$

   The relative error of this approximation is less than $2^{-57.2}$.
   The coefficients of this formula were derived by transforming the minimax rational approximation (in relative error, over the range $0 \leqq x^2 \leqq 1/4$) for $\arcsin(x)/x$ of the following form:

   $$\frac{\arcsin(x)}{x} \cong a_0 + x^2 \left[\frac{a_1 + a_2x^2 + a_3x^4 + a_4x^6 + a_5x^8}{b_0 + b_1x^2 + b_2x^4 + b_3x^6 + x^8} \right].$$

   Minimax was taken under the constraint that $a_0 = 1$ exactly.
   If $0 \leqq x \leqq 1/2$, $\arccos(x)$ is computed as:

   $$\arccos(x) = \frac{\pi}{2} - \arcsin(x).$$

2. If $1/2 < x \leqq 1$, then compute $\arccos(x)$ essentially as:

   $$\arccos(x) = 2 \cdot \arcsin\left(\sqrt{\frac{1-x}{2}}\right).$$

   This case is now reduced to the first case because within these limits,

   $$0 \leqq \sqrt{\frac{1-x}{2}} \leqq 1/2.$$

   This computation uses the real square root subprogram (DSQRT).
   If $1/2 < x \leqq 1$, $\arcsin(x)$ is computed as:

   $$\arcsin(x) = \frac{\pi}{2} - \arccos(x).$$

   Implementation of the above algorithms (steps 1 and 2) was carried out with care to minimize the round-off errors.

3. If $-1 \leqq x < 0$, then $\arcsin(x) = -\arcsin|x|$, and $\arccos(x) = \pi - \arccos|x|$. This reduces these cases to one of the two positive cases.

Effect of an Argument Error

$E \sim \frac{\Delta}{\sqrt{1-x^2}}$. For small values of x, $E \sim \Delta$. Toward the limits (± 1) of the range a small Δ causes a substantial error in the answer. For the arcsine, $\epsilon \sim \delta$ if the value of x is small.

QARSIN/QARCOS

Algorithm

1. If $0 \leq x \leq \frac{1}{2}$, then compute arcsin (x) by a minimax rational approximation of the following form:

   $w = 2x^2$, and

   $$\arcsin(x) \cong x + x \cdot w\left[\frac{a_0 + w[a_1 + a_2 w + \ldots + a_9 w^8]}{b_0 + b_1 w + \ldots + b_8 w^8 + w^9}\right]$$

   Coefficients $\{a_i, b_i\}$ were determined by a minimax technique and the relative error of this approximation is less than 16^{-28}. The order of evaluating this rational form was so chosen as to reduce round-off errors.

   If $0 \leq x \leq \frac{1}{2}$, arccos (x) is computed as:

   $$\arccos(x) = \frac{\pi}{2} - \arcsin(x).$$

2. If $\frac{1}{2} < x \leq 1$, then compute arccos (x) essentially as:

   $$\arccos(x) = 2 \cdot \arcsin\left(\sqrt{\frac{1-x}{2}}\right)$$

   Or more specifically, $w = 1 - x$, $z = \sqrt{2(1-x)}$, and arccos $(x) \cong z + z \quad w\,[a_0 +$ the above rational form$]$.

This case is now reduced to the first case because, within these limits,

$$0 \leq \sqrt{\frac{1-x}{2}} \leq \frac{1}{2}.$$

   This computation uses the square root subroutine (QSQRT).

   If $\frac{1}{2} < x \leqslant 1$, arcsin (x) is computed as:

   $$\arcsin(x) = \frac{\pi}{2} - \arccos(x).$$

3. If $-1 \leq x < 0$, then $\arcsin(x) = -\arcsin|x|$ and $\arccos(x) = \pi - \arccos|x|$.

   This reduces these cases to one of the two positive cases.

Effect of an Argument Error

$E \sim \frac{\Delta}{\sqrt{1-x^2}}$. For a small value of x, $E \sim \Delta$. Towards the limits (± 1) of the range, a small Δ causes a substantial error in the answer. For arcsin, $\epsilon \sim \delta$ if the value of x is small.

Arctangent Subprograms

ATAN

Algorithm

1. Reduce the computation of arctan (x) to the case $0 \leqq x \leqq 1$, by using

   $$\arctan(-x) = -\arctan(x), \text{ or}$$

   $$\arctan\left(\frac{1}{|x|}\right) = \frac{\pi}{2} - \arctan|x|.$$

2. If necessary, reduce the computation further to the case $|x| \leqq \tan 15°$ by using

   $$\arctan(x) = 30° + \arctan\left(\frac{\sqrt{3} \cdot x - 1}{x + \sqrt{3}}\right).$$

   The value of $\left|\frac{\sqrt{3} \cdot x - 1}{x + \sqrt{3}}\right| \leqq \tan 15°$ if the value of x is within the range, $\tan 15° < x \leqq 1$. The value of $(\sqrt{3} \cdot x - 1)$ is computed as $(\sqrt{3} - 1)\,x - 1 + x$ to avoid the loss of significant digits.

3. For $|x| \leqq \tan 15°$, use the approximation formula:

   $$\frac{\arctan(x)}{x} \cong 0.60310579 - 0.05160454x^2 + \frac{0.55913709}{x^2 + 1.4087812}.$$

   This formula has a relative error less than $2^{-27.1}$ and can be obtained by transforming the continued fraction

   $$\frac{\arctan(x)}{x} = 1 - \frac{x^2}{3 +} \frac{\frac{x^2}{5}}{\left(\frac{5}{7} + x^{-2}\right) - w}$$

   where w has an approximate value of $\left(-\frac{75}{77}x^{-2} + \frac{3375}{77}\right) 10^{-4}$, but the true value of w is $\dfrac{\frac{4 \cdot 5}{7 \cdot 7 \cdot 9}}{\left(\frac{43}{7 \cdot 11} + x^{-2}\right) +} \cdots$.

   The original continued fraction can be obtained by transforming the Taylor series into continued fraction form.

Effect of an Argument Error

$E \sim \frac{\Delta}{1 + x^2}$. For small values of x, $\epsilon \sim \delta$; as the value of x increases, the effect of δ upon ϵ diminishes.

ATAN/ATAN2

Algorithm

1. For arctan (x_1, x_2):
   If $x_1 < 0$, use the identity $\arctan(x_1, x_2) = -\arctan(-x_1, x_2)$.
   Hence we may assume that $x_1 \geqq 0$. Then:

   If either $x_2 = 0$ or $\left|\frac{x_1}{x_2}\right| > 2^{24}$, the answer $= \frac{\pi}{2}$.

   If $x_2 < 0$ and $\left|\frac{x_1}{x_2}\right| < 2^{-24}$, the answer $= \pi$.

For the general case, if $x_2 > 0$, the answer = arctan $\left(\left|\frac{x_1}{x_2}\right|\right)$, and

if $x_2 < 0$, the answer = π − arctan $\left(\left|\frac{x_1}{x_2}\right|\right)$.

2. The computation of arctan $\left(\left|\frac{x_1}{x_2}\right|\right)$ above, or of arctan(x) for the single argument case, follows the algorithm given for the subprogram ATAN with a single argument.

Effect of an Argument Error

$E \sim \frac{\Delta}{1 + x^2}$. For small values of x, $\epsilon \sim \delta$, and as the value of x increases, the effect of ϵ upon δ diminishes.

DATAN

Algorithm

1. Reduce the computation of arctan (x) to the case $0 \leqq x \leqq 1$ by using

$$\arctan(-x) = -\arctan(x) \text{ and}$$

$$\arctan \frac{1}{|x|} = \frac{\pi}{2} - \arctan |x|.$$

2. If necessary, reduce the computation further to the case $|x| \leqq \tan 15°$ by using

$$\arctan(x) = 30° + \arctan \left(\frac{\sqrt{3} \cdot x - 1}{x + \sqrt{3}}\right).$$

The value of $\left|\frac{\sqrt{3} \cdot x - 1}{x + \sqrt{3}}\right| \leqq \tan 15°$, if the value of x is within the range tan $15° < x \leqq 1$. The value of $(\sqrt{3} \cdot x - 1)$ is computed as $(\sqrt{3} - 1)\, x - 1 + x$ to avoid the loss of significant digits.

3. For $|x| \leqq \tan 15°$, use a continued fraction of the form:

$$\frac{\arctan(x)}{x} \cong 1 + x^2\left[b_0 - \frac{a_1}{(b_1 + x^2) -} \frac{a_2}{(b_2 + x^2) -} \frac{a_3}{(b_3 + x^2)}\right].$$

The relative error of this approximation is less than $2^{-60.7}$.
The coefficients of this formula were derived by transforming a minimax rational approximation (in relative error, over the range $0 \leqq x^2 \leqq 0.071797$) for $\arctan(x)/x$ of the following form:

$$\frac{\arctan(x)}{x} \cong a_0 + x^2\left[\frac{c_0 + c_1x^2 + c_2x^4 + c_3x^6}{d_0 + d_1x^2 + d_2x^4 + x^6}\right].$$

Minimax was taken under the constraint that $a_0 = 1$ exactly.

Effect of an Argument Error

$E \sim \frac{\Delta}{1 + x^2}$. For small values of x, $\epsilon \sim \delta$, and as the value of x increases, the effect of ϵ upon δ diminishes.

DATAN/DATAN2

Algorithm

1. For arctan(x_1, x_2):
If $x_1 < 0$, use the identity $\arctan(x_1, x_2) = -\arctan(-x_1, x_2)$.
Hence we may assume that $x_1 \geqq 0$. Then:

If either $x_2 = 0$ or $\left|\frac{x_1}{x_2}\right| > 2^{56}$, the answer $= \frac{\pi}{2}$.

If $x_2 < 0$ and $\left|\frac{x_1}{x_2}\right| < 2^{-56}$, the answer $= \pi$.

For the general case, if $x_2 > 0$, the answer $= \arctan\left(\left|\frac{x_1}{x_2}\right|\right)$, and

if $x_2 < 0$, the answer $= \pi - \arctan\left(\left|\frac{x_1}{x_2}\right|\right)$.

2. The computation of $\arctan\left(\left|\frac{x_1}{x_2}\right|\right)$ above, or of $\arctan(x)$ for the single argument case, follows the algorithm given for the subprogram DATAN with a single argument.

Effect of an Argument Error

$E \sim \frac{\Delta}{1+x^2}$. For small values of x, $\epsilon \sim \delta$, and as the value of x increases, the effect of ϵ upon δ diminishes.

QATAN/QATAN2

Algorithm

1. For $\arctan(x)$, if $x < 0$, then $\arctan(x) = -\arctan(|x|)$. So assume $x \geqq 0$.
2. Define break points β_i, $i = 0, 1, 2, \ldots, 8$ as $\beta_i = \tan\left(\frac{2_i - 1}{32}\pi\right)$.

   Define origins θ_i to be approximately $\frac{i}{16}\pi$, $i = 0, 1, 2, \ldots, 8$ in such a way that $\tan\theta_i$ are exact short form numbers. $\theta_8 = \frac{\pi}{2}$ exactly.
3. $\beta_i \leqq x < \beta_{i+1}$ for $i = 0, 1, 2, \ldots, 7$, then use the following reduction:
   $$\arctan(x) = \theta_i + \arctan\left(\frac{x - \tan\theta_i}{1 + x\tan\theta_i}\right)$$
   If $\beta_8 \leqq x < \infty$, use the reduction:
   $$\arctan(x) = \frac{\pi}{2} + \arctan\left(\frac{-1}{x}\right).$$
   Note the quantity within the parentheses on the right is in either case within the basic range (β_0, β_1), that is, is less than $\frac{\pi}{32}$ in magnitude.
4. Within the basic range $-\frac{\pi}{32} \leqq x \leqq \frac{\pi}{32}$, a minimax approximation of the following form is used to compute $\arctan(x)$:
   $$\arctan(x) \cong x + a_1x^3 + a_2x^5 + \ldots + a_{12}x^{25}$$
   The relative error of this approximation is less than 2^{-112}.
   It is sufficient to compute the last three terms in double precision.
5. For $\arctan(x_1, x_2)$:
   If $x_1 < 0$, use the identity $\arctan(x_1, x_2) = -\arctan(|x_1|, x_2)$.
   Hence we may assume that $x_1 \geqq 0$. Then:
   if either $x_2 = 0$ or $\left|\frac{x_1}{x_2}\right| > 2^{112}$, the answer $\cong \frac{\pi}{2}$.

   If $x_2 < 0$ and $\left|\frac{x_1}{x_2}\right| < 2^{-112}$, the answer $\cong \pi$.

   For the general case, if $x_2 > 0$, the answer $= \arctan\left(\left|\frac{x_1}{x_2}\right|\right)$, and

   if $x_2 < 0$, the answer $= \pi - \arctan\left(\left|\frac{x_1}{x_2}\right|\right)$.

Here $\arctan\left(\left|\frac{x_1}{x_2}\right|\right)$ is computed as described in steps 1 through 4 above, except for the following simplification for the case $\beta_8 \leqq \left|\frac{x_1}{x_2}\right| < \infty$:

$$\arctan\left(\left|\frac{x_1}{x_2}\right|\right) = \frac{\pi}{2} + \arctan\left(\frac{-|x_2|}{|x_1|}\right).$$

This combines two needed extended precision divisions into one for this case.

Effect of an Argument Error

$E \sim \frac{\Delta}{1 + x^2}$. For a small value of x, $\epsilon \sim \delta$, and as the value of x increases, the effect of δ upon ϵ diminishes.

Error Functions Subprograms

ERF/ERFC

Algorithm

1. If $0 \leqq x \leqq 1$, then compute the error function by the following approximation:

$$\mathrm{erf}(x) \cong x(a_0 + a_1x^2 + a_2x^4 + \ldots + a_5x^{10}).$$

   The coefficients were obtained by the minimax approximation (in relative error) of $\mathrm{erf}(x)/x$ as a function of x^2 over the range $0 \leqq x^2 \leqq 1$. The relative error of this approximation is less than $2^{-24.6}$. The value of the complemented error function is computed as $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$.

2. If $1 < x < 2.040452$, then compute the complemented error function by the following approximation:

$$\mathrm{erfc}\ (x) \cong b_0 + b_1z + b_2z^2 + \ldots + b_9z^9$$

   where $z = x - T_0$ and $T_0 \cong 1.709472$. The coefficients were obtained by the minimax approximation (in absolute error) of the function $f(z) = \mathrm{erfc}(z + T_0)$ over the range $-0.709472 \leqq z \leqq 0.33098$. The absolute error of this approximation is less than $2^{-31.5}$. The limits of this range and the value of the origin T_0 were chosen to minimize the hexadecimal round-off errors. The value of the complemented error function within this range is between $\frac{1}{256}$ and 0.1573. The value of the error function is computed as $\mathrm{erf}(x) = 1 - \mathrm{erfc}(x)$.

3. If $2.040452 \leqq x < 13.306$, then compute the complemented error function by the following approximation:

$$\mathrm{erfc}(x) \cong e^{-z} \cdot F/x \text{ where } z = x^2 \text{ and}$$

$$F = c_0 + \frac{c_1 + c_2z + c_3z^2}{d_1z + d_2z^2 + z^3}.$$

   The coefficients for F were obtained by transforming a minimax rational approximation (in absolute errors, over the range $13.306^{-2} \leqq w \leqq 2.040452^{-2}$) of the function $f(w) = \mathrm{erfc}(x) \cdot x \cdot e^{x^2}$, $w = x^{-2}$, of the following form:

$$f(w) \cong \frac{a_0 + a_1w + a_2w^2 + a_3w^3}{b_0 + b_1w + w^2}.$$

   The absolute error of this approximation is less than $2^{-26.1}$. This computation uses the real exponential subprogram (EXP).

   If $2.040452 \leqq x < 3.919206$, then the error function is computed as $\mathrm{erf}(x) = 1 - \mathrm{erfc}(x)$.
   If $3.919206 \leqq x$, then the error function is $\cong 1$.

4. If $13.306 \leqq x$, then the error function is $\cong 1$, and the complemented error function is $\cong 0$ (underflow).
5. If $x < 0$, then reduce to a case involving a positive argument by the use of the following formulas:

$$\text{erf}(-x) = -\text{erf}(x), \text{ and } \text{erfc}(-x) = 2 - \text{erfc}(x).$$

Effect of an Argument Error

$E \sim e^{-x^2} \cdot \Delta$. 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, $\epsilon \sim \delta$. For the complemented error function, if the value of x is greater than 1, $\text{erfc}(x) \sim \frac{e^{-x^2}}{2x}$. Therefore, $\epsilon \sim 2x^2 \cdot \delta$. If the value of x is negative or less than 1, then $\epsilon \sim e^{-x^2} \cdot \Delta$.

DERF/DERFC

Algorithm

1. If $0 \leqq x < 1$, then compute the error function by the following approximation:

$$\text{erf}(x) \cong x(a_0 + a_1x^2 + a_2x^4 + \ldots + a_{11}x^{22}).$$

The coefficients were obtained by the minimax approximation (in relative error) of $\text{erf}(x)/x$ as a function of x^2 over the range $0 \leqq x^2 \leqq 1$. The relative error of this approximation is less than $2^{-56.9}$. The value of the complemented error function is computed as $\text{erfc}(x) = 1 - \text{erf}(x)$.
2. If $1 \leqq x < 2.040452$, then compute the complemented error function by the following approximation:

$$\text{erfc}(x) \cong b_0 + b_1z + b_2z^2 + \ldots b_{18}z^{18}$$

where $z = x - T_0$ and $T_0 \cong 1.709472$. The coefficients were obtained by the minimax approximation (in absolute error) of the function $f(z) = \text{erfc}(z + T_0)$ over the range $-0.709472 \leqq z \leqq 0.33098$. The absolute error of this approximation is less than $2^{-60.3}$. The limits of this range and the value of the origin T_0 were chosen to minimize the hexadecimal round-off errors. The value of the complemented error function within this range is between $\frac{1}{256}$ and 0.1573. The value of the error function is computed as $\text{erf}(x) = 1 - \text{erfc}(x)$.
3. If $2.040452 \leqq x < 13.306$, then compute the complemented error function by the following approximation:

$$\text{erfc}(x) \cong e^{-z} \cdot F/x \text{ where } z = x^2 \text{ and}$$

$$F = c_0 + \frac{d_1}{(z + c_1) +} \frac{d_2}{(z + c_2) +} \cdots \frac{d_6}{(z + c_6) +} \frac{d_7}{(z + c_7)}.$$

The coefficients for F were derived by transforming a minimax rational approximation (in absolute errors, over the range $13.306^{-2} \leqq w \leqq 2.040452^{-2}$) of the function $f(w) = \text{erfc}(x) \cdot x \cdot e^{x^2}$, $w = x^{-2}$, of the following form:

$$f(w) \cong \frac{a_0 + a_1w + a_2w^2 + \ldots + a_7w^7}{b_0 + b_1w + b_2w^2 + \ldots + b_6w^6 + w^7}.$$

The absolute error of this approximation is less than $2^{-57.9}$. This computation uses the real exponential subprogram (DEXP). If $2.040452 \leqq x < 6.092368$, then the error function is computed as $\text{erf}(x) = 1 - \text{erfc}(x)$.
If $6.092368 \leqq x$, then the error function is $\cong 1$.

4. If $13.306 \leqq x$, then the error function is $\cong 1$, and the complemented error function $\cong 0$ (underflow).

5. If $x < 0$, then reduce to a case involving a positive argument by the use of the following formulas:

$$\mathrm{erf}(-x) = -\mathrm{erf}(x), \text{ and } \mathrm{erfc}(-x) = 2 - \mathrm{erfc}(x).$$

Effect of an Argument Error

$E \sim e^{-x^2} \cdot \Delta$. 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, $\epsilon \sim \delta$. For the complemented error function, if the value of x is greater than 1, $\mathrm{erfc}(x) \sim \frac{e^{-x^2}}{2x}$. Therefore, $\epsilon \sim 2x^2 \cdot \delta$. If the value of x is negative or less than 1, then $\epsilon \sim e^{-x^2} \cdot \Delta$.

Algorithm

1. If $0 \leqq x \leqq 1$, then:

Write $a(z) = \frac{\sqrt{\pi}}{2x} \operatorname{erf}(x)$ where $z = x^2$

Then $x.2xa' + a = \frac{\sqrt{\pi}}{2} \frac{d}{dx} (\operatorname{erf}(x))$ where $a' = \frac{da}{dz} = e^{-x^2}$

that is $2za' + a = e^{-z}$

Then $2za'' + 3a' = -e^{-z} = -(2za' + a)$

so that $2za'' + (2z + 3)a' + a = 0$

Now integrate twice

$$2za' + 2za + a - \int_0^z a dz - A \qquad \text{where } A \text{ is a constant}$$

But if $z = 0$ then $x = 0$ and $a = 1$ so that $A = 1$

Hence

$$2a' + 2a + \frac{a-1}{z} - \frac{\int_0^z a dz}{z} = 0$$

and

$$2a + \int_0^z \left\{ 2a + \frac{a-1}{z} - \frac{\int_0^z a dz}{z} \right\} dz = B = 2$$

Now write $\tilde{a} = 1 + \sum_{i=1}^{m} a_i z^i$ as an approximation to a and solve

$$2\tilde{a} + \int_0^z \left\{ 2\tilde{a} + \frac{\tilde{a}-1}{z} - \frac{\int_0^z \tilde{a} dz}{z} \right\} d_z = \beta + \tau T^*_{m+1},$$

Where $T^*_m = \sum_{i=0}^{m} T^*_{m,i} \times x^i$ is the Chebyshev polynomial over the appropriate range.

Equating coefficients of powers of z and multiplying the coefficient f of z^i by i^2 we have:

$$3a_1 + 1 = \tau T^*_{m+1,1}$$

$$10a_2 + 3a_1 = 4 T^*_{m+1,2}$$

$$\vdots$$

$$m(2m+1)a_m + (2m-1)a_{m-1} = m^2 \tau T^*_{m+1,m}$$

$$(2m+1)a_m = (m+1)^2 \tau T^*_{m+1,m+1}$$

Then use the approximation

$$T^*_{m+1} \cong 0$$

to approximate $c_{m+1} z^{m+1}$ by a polynomial of degree m.
The equations solved in (A) are:

$$2c_0 = \frac{2}{\sqrt{\pi}} + T^*_{m+2,0}$$

$$c_0 + 2c_1 = \tau T^*_{m+2,1}$$

•
•
•

$$(2m+1)c_m + 2c_{m+1} = \tau T^*_{m+2,m+1}$$

$$(2m+3)c_{m+1} = \tau T^*_{m+2,m+1}$$

Finally:

$$\text{erfc}(x) = c(z) \cdot \frac{e^{-x^2}}{2x} \text{ where } z = 1/x^2$$

4. If $13.306 \leqq x$, then $\text{erf}(x) = 1$ and $\text{erfc}'(x) = 0$.
5. If $0 > x$, then $\text{erf}(x) = -\text{erf}(-x)$, and $\text{erfc}(x) = 2 - \text{erfc}(-x)$.

Effect of an Argument Error

$E \sim^{-x^2} \cdot \Delta$. 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, $\epsilon \sim \delta$. For the complicated error function, if the value of X is greater than 1, $\text{erfc}(x) \sim \frac{e^{-x^2}}{2x}$.

Therefore, $\epsilon \sim 2x^2 \cdot \delta$. If the value of x is negative or less than 1, then $\epsilon \sim e^{-x^2} \cdot \Delta$.

Exponential Subprograms

EXP

Algorithm

1. If $x < -180.218$, then 0 is given as the answer via floating-point underflow.
2. Otherwise, divide x by $\log_e 2$ and write

$$y = \frac{x}{\log_e 2} = 4a - b - d$$

where a and b are integers, $0 \leqq b \leqq 3$ and $0 \leqq d < 1$.

3. Compute 2^{-d} by the following fractional approximation:

$$2^{-d} \cong 1 - \cfrac{2d}{0.034657359\, d^2 + d + 9.9545948 - \cfrac{617.97227}{d^2 + 87.417497}}.$$

This formula can be obtained by transforming the Gaussian continued fraction

$$e^{-z} = 1 - \frac{z}{1+}\ \frac{z}{2-}\ \frac{z}{3+}\ \frac{z}{2-}\ \frac{z}{5+}\ \frac{z}{2-}\ \frac{z}{7+}\ \frac{z}{2}.$$

The maximum relative error of this approximation is 2^{-29}.

Multiply 2^{-d} by 2^{-b}.

Finally, add the hexadecimal exponent a to the characteristic of the answer.

Effect of an Argument Error

$\epsilon \sim \Delta$. If the magnitude of x is large, even the round-off error of the argument causes a substantial relative error in the answer because $\Delta = \delta \cdot x$.

DEXP

Algorithm

1. If $x < -180.2187$, then 0 is given as the answer via floating-point underflow.
2. Divide x by $\log_e 2$ and write
   $$x = \left(4a - b - \frac{c}{16}\right) \cdot \log_e 2 - r$$
   where a, b, and c are integers, $0 \leqq b \leqq 3$, $0 \leqq c \leqq 15$, and the remainder r is within the range $0 \leqq r < \frac{1}{16} \cdot \log_e 2$. This reduction is carried out in an extra precision to ensure accuracy. Then $e^x = 16^a \cdot 2^{-b} \cdot 2^{-c/16} \cdot e^{-r}$.
3. Compute e^{-r} by using a minimax polynomial approximation of degree 6 over the range $0 \leqq r < \frac{1}{16} \cdot \log_e 2$. In obtaining coefficients of this approximation, the minimax of relative errors was taken under the constraint that the constant term a_0 shall be exactly 1. The relative error is less than $2^{-56.87}$.
4. Multiply e^{-r} by $2^{-c/16}$. The 16 values of $2^{-c/16}$ for $0 \leqq c \leqq 15$ are included in the subprogram. Then halve the result b times.
5. Finally, add the hexadecimal exponent of a to the characteristic of the answer.

Effect of an Argument Error

$\epsilon \sim \Delta$. If the magnitude of x is large, even the round-off error of the argument causes a substantial relative error in the answer because $\Delta = \delta \cdot x$.

CEXP/CDEXP

Algorithm

The value of e^{x+iy} is computed as $e^x \cdot \cos(y) + i \cdot e^x \cdot \sin(y)$. The algorithms for both complex exponential subprograms are identical. Each subprogram uses the appropriate real exponential subprogram (EXP or DEXP) and the appropriate real sine/cosine subprogram (COS/SIN or DCOS/DSIN).

Effect of an Argument Error

The effect of an argument error depends upon the accuracy of the individual parts of the argument. If $e^{x+iy} = R \cdot e^{iH}$, then $H = y$ and $\epsilon(R) \sim \Delta(x)$.

QEXP

Algorithm

1. Basic computation is that of 2^x. For QEXP entry, multiply x by $\log_2 e$ in a 31 hexadecimal digit arithmetic, and raise the result to the power of 2.
2. Decompose x as $x = 4p - q - r$ where p is an integer, $q = 0, 1, 2$, or 3, and $0 \leqq r < 1$.
3. Find two indices i, j, $0 \leqq i \leqq 8, 0 \leqq j \leqq 3$ such that $4i + j$ is the integer nearest to $32r$.

   Using these indices, select two encoded constants α_i, β_j where
   $\alpha_i = [2^{-i/8}]$, $\beta_j = [2^{-j/32}]$.

   Here the bracket indicates rounding to the nearest 17 binary digit number. Obtain the product $\Psi_{ij} = \alpha_i \beta_j$.

4. Obtain the reduced argument $s = -r - \log_2 (\Psi_{ij})$ accurately by subtracting $\log_2 (\Psi_{ij}) = \log_2 \alpha_i + \log_2 \beta_j +$ in an extra precision. Constants $\log_2 \alpha_i$ and $\log_2 \beta_j$ are encoded in 31 hexadecimal digits of accuracy. Then s is approximately bounded by $\pm \frac{1}{64}$.
5. Compute 2^s by a minimax approximation of the form:

$$2^s \cong 1 + \frac{2sP(s^2)}{Q(s^2) - sp(s^2)}$$

   where P and Q are polynomials of degree 2.
6. Then $2^x = 16^p \cdot (2^{-q}\Psi_{ij}) \cdot 2^8$. In assembling this product a virtual rounding is applied.
7. The limited use of extra precision arithmetic in the above computation enhances accuracy of both *QEXP* and A***B* application (see note below).

Effect of an Argument Error

$\epsilon \sim \Delta$. If the magnitude of x is large, even the roundoff error of the argument causes a substantial relative error in the answer because $\Delta = \delta \cdot x$.

CQEXP

Algorithm

The value of e^{x+iy} is computed as $e^x \cdot \cos(y) + i \cdot e^x \cdot \sin(y)$. The algorithms for both complex exponential subprograms are identical. Each subprogram uses the appropriate real exponential subprogram (QEXP) and the appropriate real sine/cosine subprogram (QCOS/QSIN).

Effect of an Argument Error

The effect of the argument error depends upon the accuracy of the individual parts of the argument. If $e^{x+iy} = R \cdot e^{iH}$, then $H = y$ and $\epsilon(R) \sim \Delta(x)$.

Gamma and Log Gamma Subprograms

GAMMA/ALGAMA

Algorithm

1. If $0 < x \leqq 2^{-252}$, then compute log-gamma as $\log_e\Gamma(x) \cong -\log_e(x)$. This computation uses the real logarithm subprogram (ALOG).
2. If $2^{-252} < x < 8$, then compute log-gamma by taking the natural logarithm of the value obtained for gamma. The computation of gamma depends upon the range into which the argument falls.
3. If $2^{-252} < x < 1$, then use $\Gamma(x) = \dfrac{\Gamma(x+1)}{x}$ to reduce to the next case.
4. If $1 \leqq x \leqq 2$, then compute gamma by the minimax rational approximation (in absolute error) of the following form:

$$\Gamma(x) \cong c_0 + \frac{z\,[a_0 + a_1 z + a_2 z^2 + a_3 z^3]}{b_0 + b_1 z + b_2 z^2 + z^3}$$

   where $z = x - 1.5$. The absolute error of this approximation is less than $2^{-25.9}$.
5. If $2 < x < 8$, then use $\Gamma(x) = (x-1)\,\Gamma(x-1)$ to reduce step by step to the preceding case.
6. If $8 \leqq x$, then compute log-gamma by the use of Stirling's formula:

$$\log_e\Gamma(x) \cong x(\log_e(x) - 1) - \tfrac{1}{2}\log_e(x) + \tfrac{1}{2}\log_e(2\pi) + G(x).$$

The modifier term $G(x)$ is computed as

$$G(x) \cong d_0x^{-1} + d_1x^{-2}.$$

These coefficients were obtained by a form of minimax approximation minimizing the ratio of the absolute error to the value of x. The absolute error is less than $x \cdot 2^{-26.2}$. Remembering the fact that $x < \log_e\Gamma(x)$ in this range, the contribution of this error to the relative error of the value for log-gamma is less than $2^{-26.2}$. This computation uses the real logarithm subprogram (ALOG).

For gamma, compute $\Gamma(x) = e^y$, where y is the value obtained for log-gamma. This computation uses the real exponential subprogram (EXP).

Effect of an Argument Error

$\epsilon \sim \psi(x) \cdot \Delta$ for gamma, and $E \sim \psi(x) \cdot \Delta$ for log-gamma, where ψ is the digamma function.

If $\frac{1}{2} < x < 3$, then $-2 < \psi(x) < 1$. Therefore, $E \sim \Delta$ 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 $\psi(x) \sim \log_e(x)$. Therefore, for gamma, $\epsilon \sim \delta x \cdot \log_e(x)$. 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, $\epsilon \sim \delta$.

DGAMMA/DLGAMA

Algorithm

1. If $0 < x \leqq 2^{-252}$, then compute log-gamma as $\log_e\Gamma(x) \cong -\log_e(x)$. This computation uses the real logarithm subprogram (DLOG).
2. If $2^{-252} < x < 8$, then compute log-gamma by taking the natural logarithm of the value obtained for gamma. The computation of gamma depends upon the range into which the argument falls.
3. If $2^{-252} < x < 1$, then use $\Gamma(x) = \dfrac{\Gamma(x+1)}{x}$ to reduce to the next case.
4. If $1 \leqq x \leqq 2$, then compute gamma by the minimax rational approximation (in absolute error) of the following form:
$$\Gamma(x) \cong c_0 + \frac{z\,[a_0 + a_1z + \ldots + a_6z^6]}{b_0 + b_1z + \ldots + b_6z^6 + z^7}$$
where $z = x - 1.5$. The absolute error of this approximation is less than $2^{-59.3}$.
5. If $2 < x < 8$, then use $\Gamma(x) = (x - 1)\,\Gamma(x - 1)$ to reduce to the preceding case.
6. If $8 \leqq x$, then compute log-gamma by the use of Stirling's formula:
$$\log_e\Gamma(x) \cong x(\log_e(x) - 1) - \tfrac{1}{2}\log_e(x) + \tfrac{1}{2}\log_e(2\pi) + G(x).$$
The modifier term $G(x)$ is computed as
$$G(x) \cong d_0x^{-1} + d_1x^{-3} + d_2x^{-5} + d_3x^{-7} + d_4x^{-9}.$$
These coefficients were obtained by a form of minimax approximation minimizing the ratio of the absolute error to the value of x. The absolute error is less than $x \cdot 2^{-56.1}$. Remembering the fact that $x < \log_e\Gamma(x)$ in this range, the contribution of this error to the relative error of the value for log-gamma is less than $2^{-56.1}$. This computation uses the real logarithm subprogram (DLOG). For gamma, compute $\Gamma(x) = e^y$, where y is the value obtained for log-gamma. This computation uses the real exponential subprogram (DEXP).

Effect of an Argument Error

$\epsilon \sim \psi(x) \cdot \Delta$ for gamma, and $E \sim \psi(x) \cdot \Delta$ for log-gamma, where ψ is the digamma function.

If $\frac{1}{2} < x < 3$, then $-2 < \psi(x) < 1$. Therefore, $E \sim \Delta$ 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 $\psi(x) \sim \log_e(x)$. Therefore, for gamma, $\epsilon \sim \delta \cdot x \cdot \log_e(x)$. 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, $\epsilon \sim \delta$.

Hyperbolic Sine and Cosine Subprograms

SINH/COSH

Algorithm

1. If $|x| < 1.0$, then compute $\sinh(x)$ as:

   $\sinh(x) \cong x + c_1x^3 + c_2x^5 + c_3x^7.$

   The coefficient c_i were obtained by the minimax approximation (in relative error) of $\frac{\sinh(x)}{x}$ as a function of x^2. The maximum relative error of this approximation is $2^{-25.6}$.
2. If $x \geqq 1.0$, then $\sinh(x)$ is computed as:

   $\sinh(x) = (1 + \delta)\,[e^{x+\log_e v} - v^2/e^{x+\log_e v}].$

   Here, $1 + \delta = \frac{1}{2v}$, so that this expression is theoretically equivalent to $[e^x - e^{-x}]/2$. The value of v (and consequently those of $\log_e v$ and δ) was so chosen as to satisfy the following conditions:

   a) v is slightly less than ½, so that $\delta > 0$ and small.

   b) $\log_e v$ is an exact multiple of 2^{-16}.

   The condition *b)* ensures that the addition $x + \log_e v$ is carried out exactly. This maneuver was designed to reduce the round-off errors and also to enlarge the limits of acceptable arguments. This computation uses the real exponential subprogram (EXP).
3. If $x \leqq -1.0$, use $\sinh(x) = -\sinh(|x|)$ to reduce to case 2 above.
4. If cosh (x) is desired, then for all valid values of arguments use the identity: $\cosh(x) = (1 + \delta)\,[e^{x+\log_e v} + v^2/e^{x+\log_e v}]$. Here the notation and the consideration are identical to case 2 above. This computation uses the real exponential subprogram (EXP).

Effect of an Argument Error

For the hyperbolic sine, $E \sim \Delta \cdot \cosh(x)$ and $\epsilon \sim \Delta \cdot \coth(x)$.

For the hyperbolic cosine, $E \sim \Delta \cdot \sinh(x)$ and $\epsilon \sim \delta \cdot \tanh(x)$.
Specifically, for the cosine, $\epsilon \sim \Delta$ over the entire range; for the sine, $\epsilon \sim \delta$ for small values of x.

DSINH/DCOSH

Algorithm

1. If $|x| < 0.881374$, then compute $\sinh(x)$ as:

   $$\sinh(x) \cong c_0x + c_1x^3 + c_2x^5 + \ldots + c_6x^{13}.$$

   The coefficients c_i were obtained by the minimax approximation (in relative error) of $\frac{\sinh(x)}{x}$ as the function of x^2. Minimax was taken under the constraint that $c_0 = 1$ exactly. The maximum relative error of this approximation is $2^{-55.7}$.
2. If $x \geqq 0.881374$, then $\sinh(x)$ is computed as:

   $$\sinh(x) = (1 + \delta)\,[e^{x+\log_e v} - v^2/e^{x+\log_e v}].$$

   Here, $1 + \delta = \frac{1}{2v}$, so that this expression is theoretically equivalent to $[e^x - e^{-x}]/2$. The value of v (and consequently those of $\log_e v$ and δ) was so chosen as to satisfy the following conditions:

   a) v is slightly less than ½, so that $\delta > 0$ and small.

   b) $\log_e v$ is an exact multiple of 2^{-16}.

   The condition *b*) ensures that the addition $x + \log_e v$ is carried out exactly. This maneuver was designed to reduce the round-off errors and also to enlarge the limits of acceptable arguments. This computation uses the real exponential subprogram (DEXP).
3. If $x \leqq -0.881374$, then use $\sinh(x) = -\sinh(|x|)$ to reduce to case 2 above.
4. If $\cosh(x)$ is desired, then, for all valid arguments use the identity:
   $\cosh(x) = (1 + \delta)\,[e^{x+\log_e v} + v^2/e^{x+\log_e v}]$. Here the notation and the consideration are identical to case 2 above. This computation uses the real exponential subprogram (DEXP).

Effect of an Argument Error

For the hyperbolic sine, $E \sim \Delta \cdot \cosh(x)$ and $\epsilon \sim \Delta \cdot \coth(x)$.
For the hyperbolic cosine, $E \sim \Delta \cdot \sinh(x)$ and $\epsilon \sim \delta \cdot \tanh(x)$.
Specifically, for the cosine, $\epsilon \sim \Delta$ over the entire range; for the sine, $\epsilon \sim \delta$ for the small values of x.

QSINH/QCOSH

Algorithm

1. If $|x| < 1$ then compute $\sinh(x)$ as:

   $$\sinh(x) \cong c_0x + c_1x^3 + c_2x^5 + \ldots + c_{12}x^{25}.$$

   The coefficients c_i were obtained by the minimax approximation (in relative error) of $\frac{\sinh(x)}{x}$ as the function of x^2. Minimax was taken under the constraint that $c_0 = 1$ exactly. The maximum relative error of this approximation is less than 2^{-112}.
2. If $x \geqq 1$ then $\sinh(x)$ is computed as:

   $$\sinh(x) = (1 + \delta)\,[e^{x+\log_e v} - v^2/e^{x+\log_e v}].$$

   Here, $1 + \delta = \frac{1}{2v}$, so that this expression is theoretically equivalent to $[e^x - e^{-x}]/2$. The value of v (and consequently those of $\log_e v$ and δ) was so chosen as to satisfy the following conditions.

a) v is slightly less than $\frac{1}{2}$, so that $\delta > 0$ and small.

b) $\log_e v$ is an exact multiple of 2^{-16}.

The condition b) ensures that the addition $x + \log_e v$ is carried out exactly. This maneuver was designed to reduce the round-off errors and also to enlarge the limits of acceptable arguments. This computation uses the exponential subprogram. Accuracy of the quotient $v^2/e^{x+\log_e v}$ is not critical if x is large. For $x > 21.85$, a double precision division yields a sufficiently accurate result.

3. If $x \leqq -1$ then use $\sinh(x) = -\sinh(|x|)$ to reduce the case to 2 above.
4. If $\cosh(x)$ is desired, for all allowable arguments use the identity: $\cosh(x) = (1 + \delta)\,[e^{x+\log_e v} + v^2/e^{x+\log_e v}]$. Here the notation and the consideration are identical to the case 2 above.

Effect of an Argument Error

For hyperbolic sine, $E \sim \Delta \cdot \cosh(x)$ and $\epsilon \sim \Delta \cdot \coth(x)$. For hyperbolic cosine, $E \sim \Delta \cdot \sinh(x)$ and $\epsilon \sim \delta \cdot \tanh(x)$. In other words, for cosine, $\epsilon \sim \Delta$ over the entire range; for sine $\epsilon \sim \delta$ for small values of x.

Hyperbolic Tangent Subprograms

TANH

Algorithm

1. If $|x| \leqq 2^{-12}$, then $\tanh(x) \cong x$.
2. If $2^{-12} < |x| \leqq 0.7$, use the following fractional approximation:

$$\frac{\tanh(x)}{x} \simeq 1 - x^2\left[0.0037828 + \frac{0.8145651}{x^2 + 2.471749}\right].$$

The coefficients of this approximation were obtained by taking the minimax of relative error, over the range $x^2 < 0.49$, of approximations of this form under the constraint that the first term shall be exactly 1.0. The maximum relative error of this approximation is $2^{-26.4}$.

3. If $0.7 < x < 9.011$, then use the identity $\tanh(x) = 1 - \frac{2}{(e^x)^2 + 1}$.

The computation for this case uses the real exponential subprogram (EXP).

4. If $x \geqq 9.011$, than $\tanh(x) \cong 1$.
5. If $x < -0.7$, then use the identity $\tanh(x) = -\tanh(-x)$.

Effect of an Argument Error

$E \sim (1 - \tanh^2 x)\,\Delta$, and $\epsilon \sim \frac{2\Delta}{\sinh(2x)}$. For small values of x, $\epsilon \sim \delta$, and as the value of x increases, the effect of δ upon ϵ diminishes.

DTANH

Algorithm

1. If $|x| \leqq 2^{-28}$, then $\tanh(x) \cong x$.
2. If $2^{-28} < |x| < 0.54931$, use the following fractional approximation:

$$\frac{\tanh(x)}{x} \cong c_0 + \frac{d_1 x^2}{x^2 + c_1} + \frac{d_2}{x^2 + c_2} + \frac{d_3}{x^2 + c_3}.$$

This approximation was obtained by rewriting a minimax approximation of the following form:

$$\frac{\tanh(x)}{x} \cong c_0 + x^2 \cdot \frac{a_0 + a_1 x^2 + a_2 x^4}{b_0 + b_1 x^2 + b_2 x^4 + x^6}.$$

Here the minimax of relative error, over the range $x^2 \leqq 0.30174$, was taken under the constraint that c_0 shall be exactly 1.0. The maximum relative error of the above is 2^{-63}.

3. If $0.54931 \leqq x < 20.101$, then use the identity $\tanh(x) = 1 - \frac{2}{e^{2x} + 1}$.

   This computation uses the double precision exponential subprogram (DEXP).
4. If $x \geqq 20.101$, then $\tanh(x) \cong 1$.
5. If $x \leqq -0.54931$, then use the identity $\tanh(x) = -\tanh(-x)$.

Effect of an Argument Error

$E \sim (1 - \tanh^2 x)\,\Delta$, and $\epsilon \sim \frac{2\Delta}{\sinh(2x)}$. For small values of x, $\epsilon \sim \delta$. As the value of x increases, the effect of δ upon ϵ diminishes.

QTANH

Algorithm

1. If $|x| \leqq 0.54931$, use a minimax fractional approximation of the following form:

$$\tanh(x) \cong x + \frac{x^3(a_0 + a_1x^2 + a_2x^4 + a_3x^6 + a_4x^8)}{b_0 + b_1x^2 + b_2x^4 + b_3x^6 + b_4x^8 + x^{10}}$$

   Approximation of this form attains accuracy better than 2^{-112} for x in the above range.
2. If $0.54931 < x \leqq 39.1628$, compute $\tanh(x)$ with the aid of the exponential subroutines as follows:

$$\tanh(x) = 1 - \frac{2}{e^{2x}+1}.$$

   Here if $x > 21.14$, the division is carried out in double precision to save execution time. The quotient term is so small relative to 1 that double precision is accurate enough.
3. If $x > 39.1628$, then $\tanh(x) \cong 1$.
4. If $x \leqq -0.54931$, then use the identity $\tanh(x) = -\tanh(-x)$ to reduce the case to either 3. or 4. above.

Effect of an Argument Error

$E \sim (1 - \tanh^2 x)\,\Delta$, and $\epsilon \sim \frac{2\Delta}{\sinh(2x)}$. For small values of x, $\epsilon \sim \delta$. As the value of x increases, the effect of δ upon ϵ diminishes.

Logarithmic Subprograms (Common and Natural)

ALOG/ALOG10

Algorithm

1. Write $x = 16^p \cdot 2^{-q} \cdot m$ where p is the exponent, q is an integer, $0 \leqq q \leqq 3$, and m is within the range, $\frac{1}{2} \leqq m < 1$.
2. Define two constants, a and b (where a = base point and $2^{-b} = a$), as follows:

   If $\frac{1}{2} \leqq m < \frac{1}{\sqrt{2}}$, then $a = \frac{1}{2}$ and $b = 1$.

   If $\frac{1}{\sqrt{2}} \leqq m < 1$, then $a = 1$ and $b = 0$.
3. Write $z = \frac{m - a}{m + a}$. Then, $m = a \cdot \frac{1 + z}{1 - z}$ and $|z| < 0.1716$.
4. Now, $x = 2^{4p-q-b} \cdot \frac{1+z}{1-z}$, and $\log_e(x) = (4p - q - b)\log_e 2 + \log_e\left(\frac{1+z}{1-z}\right)$.

Logarithmic Subprograms (Common and Natural)

ALOG/ALOG10

Algorithm

1. Write $x = 16^p \cdot 2^{-q} \cdot m$ where p is the exponent, q is an integer, $0 \leqq q \leqq 3$, and m is within the range, $\frac{1}{2} \leqq m < 1$.
2. Define two constants, a and b (where a = base point and $2^{-b} = a$), as follows:

   If $\frac{1}{2} \leqq m < \frac{1}{\sqrt{2}}$, then $a = \frac{1}{2}$ and $b = 1$.

   If $\frac{1}{\sqrt{2}} \leqq m < 1$, then $a = 1$ and $b = 0$.
3. Write $z = \frac{m - a}{m + a}$. Then, $m = a \cdot \frac{1 + z}{1 - z}$ and $|z| < 0.1716$.
4. Now, $x = 2^{4p-q-b} \cdot \frac{1 + z}{1 - z}$, and $\log_e(x) = (4p - q - b) \log_e 2 + \log_e\left(\frac{1 + z}{1 - z}\right)$.

DLOG/DLOG10

Algorithm

1. Write $x = 16^p \cdot 2^{-q} \cdot m$ where p is the exponent, q is an integer, $0 \leqq q \leqq 3$, and m is within the range $\frac{1}{2} \leqq m < 1$.
2. Define two constants, a and b (where a = base point and $2^{-b} = a$), as follows:

   If $\frac{1}{2} \leqq m < \frac{1}{\sqrt{2}}$, then $a = \frac{1}{2}$ and $b = 1$.

   If $\frac{1}{\sqrt{2}} \leqq m < 1$, then $a = 1$ and $b = 0$.
3. Write $z = \frac{m-a}{m+a}$. Then, $m = a \cdot \frac{1+z}{1-z}$ and $|z| < 0.1716$.
4. Now, $x = 2^{4p-q-b} \cdot \frac{1+z}{1-z}$, and $\log_e x = (4p - q - b) \log_e 2 + \log_e \left(\frac{1+z}{1-z}\right)$.
5. To obtain $\log_e\left(\frac{1+z}{1-z}\right)$, first compute $w = 2z = \frac{m-a}{0.5m + 0.5a}$ (which is represented with slightly more significant digits than z itself), and apply an approximation of the following form:

   $$\log_e \left(\frac{1+z}{1-z}\right) \cong w \left[c_0 + c_1 w^2 \left(w^2 + c_2 + \frac{c_3}{w^2 + c_4 + \frac{c_5}{w^2 + c_6}} \right) \right].$$

   These coefficients were obtained by the minimax rational approximation of $\frac{1}{2z} \log_e \left(\frac{1+z}{1-z}\right)$ over the range $z^2 \epsilon (0, 0.02944)$ under the constraint that c_0 shall be exactly 1.0. The maximum relative error of this approximation is less than $2^{-60.55}$.
6. If the common logarithm is desired, then $\log_{10} x = \log_{10} e \cdot \log_e x$.

Effect of an Argument Error

$E \sim \delta$. 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

Algorithm

1. Write $\log_e (x + iy) = a + ib$.
2. Then, $a = \log_e |x + iy|$ and $b =$ the principal value of arctan (y, x).
3. $\log_e |x + iy|$ is computed as follows:
   Let $v_1 = \max(|x|, |y|)$, and $v_2 = \min(|x|, |y|)$.

   Let t be the exponent of v_1, i.e., $v_1 = m \cdot 16^t$, $\frac{1}{16} \leqq m < 1$.

   Finally, let $t_1 = \begin{Bmatrix} t \text{ if } t \leqq 0 \\ t - 1 \text{ if } t > 0 \end{Bmatrix}$,
   and $s = 16^{t_1}$.

   Then, $\log_e |x + iy| = 4t_1 \cdot \log_e(2) + ½ \log_e \left[\left(\frac{v_1}{s}\right)^2 + \left(\frac{v_2}{s}\right)^2 \right]$.

Computation of v_1/s and v_2/s are carried out by manipulation of the characteristics of v_1 *and* v_2. In particular, if v_2/s is very small, it is taken to be 0. The algorithms for both complex logarithm subprograms are identical. Each subprogram uses the appropriate real natural logarithm subprogram (ALOC or DLOC) and the appropriate arctangent subprogram (ATAN2 or DATAN2).

Effect of an Argument Error

The effect of an argument error depends upon the accuracy of the individual parts of the argument. If $x + iy = r \cdot e^{ih}$ and $\log_e (x + iy) = a + ib$, then $h = b$ and $E(a) = \delta(r)$.

QLOG/QLOG10

Algorithm

1. Decompose x as $x = 16^p \cdot 2^{-q} \cdot m$, where $\frac{1}{2} \leqq m < 1$.
2. Make an estimate of $\log_2 m$ and define three indices $0 \leqq i \leqq 8$, $0 \leqq i \leqq 3$, $0 \leqq k \leqq 4$ so that $20i + 5i + k$ is the nearest integer to $-160 \cdot \log_2 m$. Using these indices, select three constants α_i, β_j, γ_k where
   $\alpha_i = [2^{-i/8}]$, $\beta_j = [2^{-j/32}]$, $\gamma_k = [2^{-k/160}]$.
   Here the bracket indicates rounding to the nearest 17 digit binary number. Obtain the exact product $\varphi_{ijk} = \alpha_i \beta_j \gamma_k$ by use of *ME* and *MXD* instructions. The 18 short constants α_i, β_j, and γ_k are encoded in the subroutine.
3. Denote $z = (m - \varphi_{ijk})/(m + \varphi_{ijk})$.
   Compute $w = 2z/\log_e(2) = (m - \varphi_{ijk})/[0.5 \cdot \log_e(2) \cdot (m + \varphi_{ijk})]$.
   The computed w is bounded approximately by $\pm \frac{1}{320}$, and it has 112 bit accuracy.
4. Compute $\log_2 \left(\frac{1 + z}{1 - z}\right) = \log_2(m) - \log_2(\varphi_{ijk})$ as follows:

   $$\log_2 \left(\frac{1 + z}{1 - z}\right) \cong w + a_1 w^3 + a_2 w^5 + \ldots + a_5 w^{11}$$

   where coefficients $\{a_n\}$ have been obtained by the minimax technique. $\log_2 \left(\frac{1 + z}{1 - z}\right)$ is approximately bounded by $\pm \frac{1}{300}$. This value is computed with full 28 hexadecimal digit accuracy, and the absolute error is at most 16^{-30}.
5. Now $\log_2(x) = 4p - q + \log_2 \alpha_i + \log_2 \beta_j + \log_2 \gamma_k + \log_2 \left(\frac{1 + z}{1 - z}\right)$.
   $\log_2 \alpha_i$, $\log_2 \beta_j$, and $\log_2 \gamma_k$ are encoded with 31 hexadecimal digits of accuracy. Combine these components in such a way that the maximum absolute error is

still 16^{-30} approximately. This is done to improve accuracy of A**B application (see Note below).

6. Truncate $\log_2(x)$ at the 28th hexadecimal digit, and multiply by $\log_e(2)$ or by $\log_{10}(2)$ to obtain $\log_e(x)$ or $\log_{10}(x)$ as desired.

Effect of an Argument Error

$E \sim \delta$. 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.

LOG/LOG 10

Algorithm

If x is R*4, then LOG (x) = ALOG(x) and LOG10(x) = ALOG1O(x).
If x is R*8, then LOG(x) = DLOG(x) and LOG10(x) = DLOG10(x).
If x is R*16, then LOG(x) = QLOG(x) and LOG10(x) = QLOG10(x).

CQLOG

Algorithm

1. Write $\log_e(x + iy) = a + ib$
2. Then, $a = \log_e |x + iy|$ and $b =$ the principal value of $\arctan(y,x)$.
3. $\log_e |x + iy|$ is computed as follows:

   Let $v_1 = \max(|x|, |y|)$, and $v_2 = \min(|x|, |y|)$.

   Let t be the exponent of v_1, i.e., $v_1 = m \cdot 16^t$, $\frac{1}{16} \leqq m < 1$.

   Finally, let $t_1 = \begin{cases} t & \text{if } t \leqq 0 \\ t - 1 & \text{if } t > 0, \end{cases}$
   and $s = 16^{t_1}$.

   Then, $\log_e |x + iy| = 4t_1 \cdot \log_e(2) + \frac{1}{2} \log_e\left[\left(\frac{v_1}{s}\right)^2 + \left(\frac{v_2}{s}\right)^2\right]$.

   Computation of v_1/s and v_2/s are carried out by manipulation of the characteristics of v_1 and v_2. In particular, if v_2/s is very small, it is given the exponent of -16 to avoid characteristic wrap-around.

Effect of an Argument Error

$E \sim \delta$. 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 Subprograms

SIN/COS

Algorithm

1. Define $z = \frac{4}{\pi} \cdot |x|$ and separate z into its integer part (q) and its fraction part (r). Then $z = q + r$, and $|x| = \left(\frac{\pi}{4} \cdot q\right) + \left(\frac{\pi}{4} \cdot r\right)$.
2. If the cosine is desired, add 2 to q. If the sine is desired and if x is negative, add 4 to q. This adjustment of q reduces the general case to the computation of $\sin(x)$ for $x \geqq 0$ because

   $$\cos(\pm x) = \sin\left(\frac{\pi}{2} + x\right), \text{ and}$$
   $$\sin(-x) = \sin(\pi + x).$$

3. Let $q_0 \equiv q \bmod 8$.

   Then, for $q_0 = 0$, $\sin(x) = \sin\left(\frac{\pi}{4} \cdot r\right)$,

   $q_0 = 1$, $\sin(x) = \cos\left(\frac{\pi}{4}(1 - r)\right)$,

   $q_0 = 2$, $\sin(x) = \cos\left(\frac{\pi}{4} \cdot r\right)$,

   $q_0 = 3$, $\sin(x) = \sin\left(\frac{\pi}{4}(1 - r)\right)$,

   $q_0 = 4$, $\sin(x) = -\sin\left(\frac{\pi}{4} \cdot r\right)$,

   $q_0 = 5$, $\sin(x) = -\cos\left(\frac{\pi}{4}(1 - r)\right)$,

   $q_0 = 6$, $\sin(x) = -\cos\left(\frac{\pi}{4} \cdot r\right)$,

   $q_0 = 7$, $\sin(x) = -\sin\left(\frac{\pi}{4}(1 - r)\right)$.

   These formulas reduce each case to the computation of either $\sin\left(\frac{\pi}{4} \cdot r_1\right)$ or $\cos\left(\frac{\pi}{4} \cdot r_1\right)$ where r_1 is either r or $(1 - r)$ and is within the range, $0 \leqq r_1 \leqq 1$.

4. If $\sin\left(\frac{\pi}{4} \cdot r_1\right)$ is needed, it is computed by a polynomial of the following form:

   $$\sin\left(\frac{\pi}{4} \cdot r_1\right) \cong r_1 (a_0 + a_1 r_1^2 + a_2 r_1^4 + a_3 r_1^6).$$

   The coefficients were obtained by interpolation at the roots of the Chebyshev polynomial of degree 4. The relative error is less than $2^{-28.1}$ for the range.

5. If $\cos\left(\frac{\pi}{4} \cdot r_1\right)$ is needed, it is computed by a polynomial of the following form:

   $$\cos\left(\frac{\pi}{4} \cdot r_1\right) \cong 1 + b_1 r_1^2 + b_2 r_1^4 + b_3 r_1^6.$$

nomial of one higher degree. The maximum relative error of the sine polynomial is 2^{-58} and that of the cosine polynomial is $2^{-64.3}$.

Effect of an Argument Error

$E \sim \Delta$. As the value of the argument increases, Δ increases. Because the function value diminishes periodically, no consistent relative error control can be maintained outside of the principal range, $-\frac{\pi}{2} \leqq x \leqq +\frac{\pi}{2}$.

CSIN/CCOS

Algorithm

1. If the sine is desired, then

$$\sin(x + iy) = \sin(x) \cdot \cosh(y) + i \cdot \cos(x) \cdot \sinh(y).$$

If the cosine is desired, then

$$\cos(x + iy) = \cos(x) \cdot \cosh(y) - i \cdot \sin(x) \cdot \sinh(y).$$

2. The value of $\sinh(x)$ is computed within the subprogram as follows. Assume $x \geqq 0$ for this, since $\sinh(-x) = -\sinh(x)$.
3. If $x \geqq 0.346574$, then use $\sinh(x) = \frac{1}{2}\left(e^x - \frac{1}{e^x}\right)$.
4. If $0 \leqq x < 0.346574$, then compute $\sinh(x)$ by use of a polynomial:

$$\frac{\sinh(x)}{x} \cong a_0 + a_1x^2 + a_2x^4.$$

The coefficients were obtained by the minimax approximation (in relative error) of $\sinh(x)/x$ over the range $0 \leqq x^2 \leqq 0.12011$ under the constraint that a_0 shall be exactly 1.0. The relative error of this approximation is less than $2^{-26.18}$.

5. The value of $\cosh(x)$ is computed as $\cosh(x) = \sinh|x| + \frac{1}{e^{|x|}}$.

This computation uses the real exponential subprogram (EXP) and the real sine/cosine subprogram (SIN/COS).

Effect of an Argument Error

To understand the effect of an argument error upon the accuracy of the answer, the programmer must understand the effect of an argument in the SIN/COS, EXP, and SINH/COSH subprograms.

CDSIN/CDCOS

Algorithm

1. If the sine is desired, then

$$\sin(x+iy) = \sin(x) \cdot \cosh(y) + i \cdot \cos(x) \cdot \sinh(y).$$

If the cosine is desired, then

$$\cos(x + iy) = \cos(x) \cdot \cosh(y) - i \cdot \sin(x) \cdot \sinh(y).$$

2. The value of $\sinh(x)$ is computed within the subprogram as follows. Assume $x \geqq 0$ for this, since $\sinh(-x) = -\sinh(x)$.
3. If $x \geqq 0.481212$, then use $\sinh(x) = \frac{1}{2}\left(e^x - \frac{1}{e^x}\right)$.
4. If $0 \leqq x < 0.481212$, then compute $\sinh(x)$ by use of a polynomial:

$$\frac{\sinh(x)}{x} \cong a_0 + a_1x^2 + a_2x^4 + a_3x^6 + a_4x^8 + a_5x^{10}.$$

The coefficients were obtained by the minimax approximation (in relative error) of $\sinh(x)/x$ over the range $0 \leqq x^2 \leqq 0.23156$ under the constraint that a_0 shall be exactly 1.0. The relative error of this approximation is less than $2^{-56.07}$.

5. The value of $\cosh(x)$ is computed as $\cosh(x) = \sinh|x| + \frac{1}{e^{|x|}}$.

   This computation uses the real exponential subprogram (DEXP) and the real sine/cosine subprogram (DSIN/DCOS).

Effect of an Argument Error

To understand the effect of an argument error upon the accuracy of the answer, the programmer must understand the effect of an argument error in the DSIN/DCOS, DEXP, and DSINH/DCOSH subprograms.

QSIN/QCOS

Algorithm

1. Separate the argument into an integral multiple of $\frac{\pi}{2}$ and the remainder part:

   $$|x| = \frac{\pi}{2} \cdot q + r \text{ where } q \text{ is an integer, and } -\frac{\pi}{4} \leqq r < \frac{\pi}{4}.$$

   In this decomposition, after q is estimated in the working precision, r is accurately computed as $r = |x| - \frac{\pi}{2} \cdot q$ with the aid of approximately 10 hexadecimal guard digits.

2. Add 1 to q if cosine is desired, since $\cos(\pm x) = \sin\left(\frac{\pi}{2} + x\right)$.

   Add 2 to q if sine is desired and x is negative, since $\sin(-x) = \sin(\pi + x)$. These adjustments reduce the general case to computation of $\sin(x)$ for $x \geqq 0$.

3. Let $q_0 \equiv q \bmod 4$. Then,

   if $q_0 = 0, \sin(|x|) = \sin(r)$
   $q_0 = 1, \sin(|x|) = \cos(r)$
   $q_0 = 2, \sin(|x|) = -\sin(r)$
   $q_0 = 3, \sin(|x|) = -\cos(r)$

4. Compute $\sin(r)$ or $\cos(r)$ as follows:

   $\sin(r) \cong r + a_1r^3 + a_2r^5 + \ldots + a_{11}r^{23}$
   $\cos(r) \cong 1 + b_1r^2 + b_2r^4 + \ldots + b_{12}r^{24}$

   Coefficients $\{a_j\}$, $\{b_j\}$ are determined by the minimax technique as applied to the range $0 \leqq r \leqq \frac{\pi}{4}$. The relative errors of these approximations are less than 2^{-112}.

Effect of an Argument Error

$E \sim \Delta$. As the value of x increases, Δ increases. Because the function value diminishes periodically, no consistent relative error control can be normally maintained outside the principal range $-\frac{\pi}{2} \leqq x \leqq +\frac{\pi}{2}$.

CQSIN/CQCOS

Algorithm

1. If the sine is desired, then

$$\sin(x + iy) = \sin(x) \cdot \cosh(y) + i \cdot \cos(x) \cdot \sinh(y).$$

   If the cosine is desired, then

$$\cos(x + iy) = \cos(x) \cdot \cosh(y) - i \cdot \sin(x) \cdot \sinh(y).$$

2. The value of $\sinh(x)$ is computed within the subprogram as follows. Assume $x \geqq 0$ for this, since $\sinh(-x) = -\sinh(x)$.

3. If $x \geqq 0.481212$, then use $\sinh(x) = \frac{1}{2}\left(e^x - \frac{1}{e^x}\right)$.

4. If $0 \leqq x < 0.481212$, then compute $\sinh(x)$ by the use of the polynomial:

$$\frac{\sinh(x)}{x} \cong a_0 + a_1x^2 + a_2x^4 + \ldots + a_{10}x^{20}$$

   The coefficients were obtained by the minimax approximation (in relative error) of $\sinh(x)/x$ over the range $0 \leqq x^2 \leqq 0.23156$ under the constraint that a_0 shall be exactly 1.0. The relative error of this approximation is less than 2^{-112}. The highest three terms of this polynomial need only be evaluated in double precision.

5. The value of $\cosh(x)$ is computed as $\cosh(x) = \sinh|x| + \frac{1}{e^{|x|}}$.

Effect of an Argument Error

Combine such effects on sine/cosine/hyperbolic-sine/hyperbolic-cosine functions according to the formula in step 1 of the algorithm.

Square Root Subprograms

SQRT

Algorithm

1. If $x = 0$, then the answer is 0.
2. Write $x = 16^{2p-q} \cdot m$, where $2p - q$ is the exponent and q equals either 0 or 1; m is the mantissa and is within the range $\frac{1}{16} \leqq m < 1$.
3. Then, $\sqrt{x} = 16^p \cdot 4^{-q}\sqrt{m}$.
4. For the first approximation of $\sqrt{x}$, compute the following:

   $$y_0 = 16^p \cdot 4^{-q} \cdot \left(1.681595 - \frac{1.288973}{0.8408065 + m}\right).$$

   This approximation attains the minimax relative error for hyperbolic fits of $\sqrt{x}$. The maximum relative error is $2^{-5.748}$.
5. Apply the Newton-Raphson iteration

   $$y_{n+1} = ½\left(y_n + \frac{x}{y_n}\right)$$

   twice. The second iteration is performed as

   $$y_2 = ½\left(y_1 - \frac{x}{y_1}\right) + \frac{x}{y_1},$$

   with a partial rounding. The maximum relative error of y_2 is theoretically $2^{-25.9}$.

Effect of an Argument Error

$$\epsilon \sim \frac{1}{2}\,\delta.$$

DSQRT

Algorithm

1. If $x = 0$, then the answer is 0.
2. Write $x = 16^{2p-q} \cdot m$, where $2p - q$ is the exponent and q equals either 0 or 1; m is the mantissa and is within the range $\frac{1}{16} \leqq m < 1$.
3. Then, $\sqrt{x} = 16^p \cdot 4^{-q}\,\sqrt{m}$.
4. For the first approximation of $\sqrt{x}$, compute the following:

   $$y_0 = 16^p \cdot 4^{1-q} \cdot 0.2202\,(m + 0.2587).$$

   The extrema of relative errors of this approximation for $q = 0$ are $2^{-3.202}$ at $m = 1$, $2^{-3.265}$ at $m = 0.2587$, and $2^{-2.925}$ at $m = \frac{1}{16}$. This approximation, rather than the minimax approximation, was chosen so that the quantity $\frac{x}{y_3} - y_3$ below becomes less than 16^{p-8} in magnitude. This arrangement allows us to substitute short form counterparts for some of the long form instructions in the final iteration.

5. Apply the Newton Raphson iteration

$$y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right)$$

four times to y_0, twice in the short form and twice in the long form. The final step is performed as

$$y_4 = y_3 + \frac{1}{2}\left(\frac{x}{y_3} - y_3\right)$$

with an appropriate truncation maneuver to obtain a virtual rounding. The maximum relative error of the final result is theoretically $2^{-63.23}$.

Effect of an Argument Error

$$\epsilon \sim \frac{1}{2}\,\delta$$

CSQRT/CDSQRT

Algorithm

1. Write $\sqrt{x + iy} = a + ib$.
2. Compute the value $z = \sqrt{\dfrac{|x| + |x + iy|}{2}}$ as $k \cdot \sqrt{w_1 + w_2}$ where k, w_1 and w_2 are defined in 3 or 4, below. In any case let $v_1 = \max(|x|, |y|)$ and $v_2 = \min(|x|, |y|)$.
3. In the special case when either $v_2 = 0$ or v_1 greatly exceeds v_2, let $\omega_1 = v_2$ and $\omega_2 = v_1$ so that $\omega_1 + \omega_2$ is effectively equal to v_1.
   Also let $k = 1$ if $v_1 = |x|$ and
   $k = 1/\sqrt{2}$ if $v_1 = |y|$.
4. In the general case, compute $F = \sqrt{\frac{1}{4} + \frac{1}{4}\left(\dfrac{v_2}{v_1}\right)^2}$.
   If $|x|$ is near the underflow threshold, then take
   $w_1 = |x|$, $w_2 = v_1 \cdot 2F$, and $k = 1/\sqrt{2}$.
   If $v_1 \cdot F$ is near the overflow threshold, then take
   $w_1 = |x|/4$, $w_2 = v_1 \cdot F/2$, and $k = \sqrt{2}$.
   In all other cases, take $w_1 = |x|/2$, $w_2 = v_1 \cdot F$, and $k = 1$.
5. If $z = 0$, then $a = 0$ and $b = 0$.
   If $z \neq 0$ and $x \geqq 0$, then $a = z$, and
   $$b = \frac{y}{2z}.$$
   If $z \neq 0$ and $x < 0$, then $a = \left|\dfrac{y}{2z}\right|$, and
   $$b = (\text{sign}\, y) \cdot z.$$

The algorithms for both complex square root subprograms are identical. Each subprogram uses the appropriate real square root subprogram (SQRT or DSQRT).

Effect of an Argument Error

The effect of an argument error depends upon the accuracy of the individual parts of the argument. If $x + iy = r \cdot e^{ih}$ and $\sqrt{x + iy} = R \cdot e^{iH}$,

then $\epsilon(R) \sim \frac{1}{2} \delta(r)$, and $\epsilon(H) \sim \delta(h)$.

QSQRT

Algorithm

1. Let $x = 16^{2p+q} \cdot m$, where p is an integer, $q = 0$ or 1, and $\frac{1}{16} \leqq m < 1$. Let $x_1 = 16^{32-q} \cdot m$

   This scaling by 16^{32} is made to avoid intermediate underflows.

2. Compute the first approximation y_0 to $\sqrt{x_1}$ as follows:

   $$y_0 = 16^{16} \cdot 4^{-q} \left\{ 1.807018 - \frac{1.576942}{0.9540356 + m} \right\}$$

   These coefficients were determined to minimize the relative error of the approximation while being exact at $m = 1$. The maximum relative error is $2^{-5.48}$.

3. Apply Newton Raphson iteration three times – twice in short form and once in long form.

   $$y_i = \frac{1}{2}\left(y_{i-1} + \frac{x_1}{y_{i-1}} \right) \qquad i = 1, 2, 3.$$

   At the end of the third iteration, the relative error ϵ_3 of y_3 is at most 2^{-41}.

4. Apply to y_3 the following cubic refinement in extended precision:

   $$y_4 = y_3 - 2y_3 \cdot \frac{y_3^2 - x_1}{3y_3^2 + x_1}.$$

   The relative error ϵ_4 of y_4 is $\frac{1}{4}(\epsilon_3)^3$ or 2^{-125}.

   Since the right hand term is only a correctional term, a simplified extended division suffices. In the process of assembling y_4, a virtual rounding is given.

5. Replace the exponent of y_4 with the correct exponent $p + q$.

Effect of an Argument Error

$$\epsilon \sim \frac{1}{2} \delta$$

CQSQRT

Algorithm

1. Write $\sqrt{x + iy} = a + ib$
2. Let $16^{2p+q-1} \leqq \max(|x|, |y|) < 16^{2p+q}$, $q = 0$, or 1
   Let $x_1 = x \cdot 16^{-2p}$, and $y_1 = y \cdot 16^{-2p}$.
   This scaling operation is carried out by manipulation of the characteristic fields of x and y. In doing this necessary precaution is exercised to avoid the anomaly of characteristic wrap-around.
3. Compute $z_1 = \sqrt{\dfrac{|x_1| + |x_1 + iy_1|}{2}}$
   Restore scaling: $z = 16^p \cdot z_1$
4. If $z = 0$, then $a = 0$ and $b = 0$.
   If $z \neq 0$, and $x \geqq 0$, then $a = z$, and
   $$b = \frac{y}{2z}.$$
   If $z \neq 0$ and $x < 0$, then $a = \left|\dfrac{y}{2z}\right|$, and
   $$b = (\text{sign } y) \cdot z.$$

Effect of an Argument Error

Using polar coordinate, write $x + iy = r \cdot e^{ih}$ and $\sqrt{x + iy} = R \cdot e^{iH}$.

Then $\epsilon(R) \sim \frac{1}{2}\delta(r)$, and $\epsilon(H) \sim \delta(h)$.

Tangent and Cotangent Subprograms

TAN/COTAN

Algorithm

1. Divide $|x|$ by $\frac{\pi}{4}$ and separate the result into the integer part (q) and the fraction part (r). Then $|x| = \frac{\pi}{4}(q + r)$.

2. Obtain the reduced argument (w) as follows:

   if q is even, then $w = r$

   if q is odd, then $w = 1 - r$.

   The range of the reduced argument is $0 \leqq w \leqq 1$.
3. Let $q_0 \equiv q \bmod 4$.

   Then for $q_0 = 0$, $\tan|x| = \tan\left(\frac{\pi}{4} \cdot w\right)$ and $\cot|x| = \cot\left(\frac{\pi}{4} \cdot w\right)$,

   $q_0 = 1$, $\tan|x| = \cot\left(\frac{\pi}{4} \cdot w\right)$ and $\cot|x| = \tan\left(\frac{\pi}{4} \cdot w\right)$,

   $q_0 = 2$, $\tan|x| = -\cot\left(\frac{\pi}{4} \cdot w\right)$ and $\cot|x| = -\tan\left(\frac{\pi}{4} \cdot w\right)$,

   $q_0 = 3$, $\tan|x| = -\tan\left(\frac{\pi}{4} \cdot w\right)$ and $\cot|x| = -\cot\left(\frac{\pi}{4} \cdot w\right)$.
4. The value of $\tan\left(\frac{\pi}{4} \cdot w\right)$ and $\cot\left(\frac{\pi}{4} \cdot w\right)$ are computed as the ratio of two polynomials:

   $$\tan\left(\frac{\pi}{4} \cdot w\right) \cong \frac{w \cdot P(u)}{Q(u)}, \quad \cot\left(\frac{\pi}{4} \cdot w\right) \cong \frac{Q(u)}{w \cdot P(u)}$$

   where $u = \frac{1}{2}w^2$ and

   $P(u) = -8.460901 + u$

   $Q(u) = -10.772754 + 5.703366 \cdot u - 0.159321 \cdot u^2$.

   These coefficients were obtained by the minimax rational approximation (in relative error) of the indicated form. The maximum relative error of this approximation is 2^{-26}. Choice of u rather than w^2 as the variable for P and Q is to improve the round-off quality of the coefficients.
5. If $x < 0$, then $\tan(x) = -\tan|x|$, and $\cot(x) = -\cot|x|$.
6. This program is provided with two kinds of error controls. One is for arguments whose magnitude is greater than $2^{18} \cdot \pi$. The other is for arguments which are very close to a singularity of the function. In either case, the precision of the argument is deemed insufficient for obtaining a reliable result. More specifically, the second control screens out the following arguments:

   a) $|x| \leqq 16^{-63}$ for COTAN (the result would overflow).

   b) x is such that one can find a singularity within eight units of the last digit value of the floating-point representation of the sum $q + r$. Singularities are cases when the cotangent ratio is to be taken and $w = 0$.

Effect of an Argument Error

$E \sim \dfrac{\Delta}{\cos^2(x)}$, and $\epsilon \sim \dfrac{2}{\sin(2x)}$ for $\tan(x)$. Therefore, near the singularities $x = \left(k + \frac{1}{2}\right)\pi$, where k is an integer, no error control can be maintained. This is also true for $\text{cotan}(x)$ for x near $k\pi$, where k is an integer.

DTAN/DCOTAN

Algorithm

1. Divide $|x|$ by $\frac{\pi}{4}$ and separate the result into integer part (q) and the fraction part (r). Then $|x| = \frac{\pi}{4}(q + r)$.
2. Obtain the reduced argument (w) as follows:
   if q is even, then $w = r$
   if q is odd, then $w = 1 - r$.

   The range of the reduced argument is $0 \leqq w \leqq 1$.
3. Let $q_0 \equiv q \bmod 4$.

   Then for $q_0 = 0$, $\tan|x| = \tan\left(\frac{\pi}{4} \cdot w\right)$ and $\cot|x| = \cot\left(\frac{\pi}{4} \cdot w\right)$,

   $q_0 = 1$, $\tan|x| = \cot\left(\frac{\pi}{4} \cdot w\right)$ and $\cot|x| = \tan\left(\frac{\pi}{4} \cdot w\right)$,

   $q_0 = 2$, $\tan|x| = -\cot\left(\frac{\pi}{4} \cdot w\right)$ and $\cot|x| = -\tan\left(\frac{\pi}{4} \cdot w\right)$,

   $q_0 = 3$, $\tan|x| = -\tan\left(\frac{\pi}{4} \cdot w\right)$ and $\cot|x| = -\cot\left(\frac{\pi}{4} \cdot w\right)$.
4. The value of $\tan\left(\frac{\pi}{4} \cdot w\right)$ and $\cot\left(\frac{\pi}{4} \cdot w\right)$ are computed as the ratio of two polynomials:

   $$\tan\left(\frac{\pi}{4} \cdot w\right) \cong \frac{w \cdot P(w^2)}{Q(w^2)}, \text{ and } \cot\left(\frac{\pi}{4} \cdot w\right) \cong \frac{Q(w^2)}{w \cdot P(w^2)}.$$

   where both P and Q are polynomials of degree 3 in w^2. The coefficients of P and Q were obtained by the minimax rational approximation (in relative error) of $\frac{1}{w} \tan\left(\frac{\pi}{4} w\right)$ of the indicated form. The maximum relative error of this approximation is $2^{-55.6}$.
5. If $x < 0$, then $\tan(x) = -\tan|x|$, and $\cot(x) = -\cot|x|$.
6. This program is provided with two kinds of error controls. One is for arguments whose magnitude is greater than $2^{50} \cdot \pi$. The other is for arguments which are very close to a singularity of the function. In either case, the precision of the argument is deemed insufficient for obtaining a reliable result. More specifically, the second control screens out the following arguments:
   a) $|x| \leqq 16^{-63}$ for COTAN (the result would overflow).
   b) x is such that one can find a singularity within eight units of the last digit value of the floating-point representation of the sum $q + r$. Singularities are cases when the cotangent ratio is to be taken and $w = 0$.

Effect of an Argument Error

$E \sim \frac{\Delta}{\cos^2(x)}$, and $\epsilon \sim \frac{2}{\sin(2x)}$ for $\tan(x)$. Therefore, near the singularities of $x = \left(k + \frac{1}{2}\right)\pi$, where k is an integer, no error control can be maintained. This is also true for $\text{cotan}(x)$ for values of x near $k\pi$, where k is an integer.

QTAN/QCOTAN

Algorithm

1. Separate argument into an integral multiple of $\frac{\pi}{2}$ and the remainder part:

   $$|x| = \frac{\pi}{2} \cdot q + r \quad \text{where } q \text{ is an integer, and } -\frac{\pi}{4} \leqq r < \frac{\pi}{4}.$$

   In this decomposition, after q is estimated in the working precision, r is accurately computed as $r = |x| - \frac{\pi}{2} \cdot q$ with the aid of approximately 10 hexadecimal guard digits.
2. If $\cot(x)$ is desired, add 1 to q, and remember to change the sign of the answer. Since $\cot(x) = -\tan\left(x + \frac{\pi}{2}\right)$, this reduces the case to computation of tangent.
3. If q is even, $\tan(|x|) = \tan(r)$, and the latter is obtained by a minimax approximation of the form:

   $$\tan(r) \cong \frac{rP(r^2)}{Q(r^2)}$$

   where P and Q are polynomials of degree 6 and 5 respectively.

   If q is odd, $\tan(|x|) = -\cot(r)$, and the latter is computed as

   $$\cot(r) \cong \frac{Q(r^2)}{rP(r^2)}$$

   using the same polynomials as the former case.

   The relative errors of these approximations are less than 2^{-111}. In evaluating these rational approximations, an exponent scaling is used to avoid intermediate partial underflows, which can result in a loss of accuracy.
4. If $x < 0$, then $\tan(x) = -\tan(|x|)$, and $\cot(x) = -\cot(|x|)$.

Effect of an Argument Error

$E \sim \frac{\Delta}{\cos^2(x)}$, and $\epsilon \sim \frac{2\Delta}{\sin(2x)}$ for $\tan(x)$. Therefore near the singularities $x = \left(k + \frac{1}{2}\right)\pi$, where k stands for integers, no error control can be maintained. This is also true for $\cot(x)$ for x near $k\pi$, where k is an integer.

Implicitly Called Subprograms

The entry point names of the following implicitly called subprograms are generated by the compiler.

Complex Multiply and Divide Subprograms

CDVD#/CMPY# (Divide/Multiply for COMPLEX*8 Arguments)

CDDVD#/CDMPY# (Divide/Multiply for COMPLEX*16 Arguments)

Algorithm

Multiply: $(A + Bi)(C + Di) = (AC - BD) + (AD + BC)i$

Divide: $(A + Bi)/(C + Di)$

1. If $|C| \leqq |D|$, set
   $A = B, B = -A, C = D, D = -C$, since
   $\frac{A + Bi}{C + Di} = \frac{B - Ai}{D - Ci}$ before step 2.
2. Set $A' = \frac{A}{C}, B' = \frac{B}{C}, D' = \frac{D}{C}$;
   then compute
   $\frac{A + Bi}{C + Di} = \frac{A' + B'i}{1 + D'i} = \frac{A' + B'D}{1 + D'D'} + \frac{B' - A'D'}{1 + D'D'}i.$

Error Conditions

Partial underflows can occur in preparing the answer.

CQMPY#/CQDVD# (Multiply/Divide for COMPLEX*32 Arguments)

Algorithm

Multiply: $(a+bi)(c+di) = (ac-bd) + (ad+bc)i$

Divide: $(a+bi)/(c+di)$

1. Let $a + bi$ and $c + di$ be the first and the second operands respectively.
2. Find exponents p_1, p_2 which satisfy the following:
   $$16^{p_1-1} \leqq \max(|a|, |b|) < 16^{p_1}, 16^{p_2-1} \leqq \max(|c|, |d|) < 16^{p_2}.$$
   Choose $q = -3$ if $p_1 \geqq 0$
   $q = 31$ if $p_1 < 0$
3. Scale c and d by 16^{p_2-q} and change sign of d if CQDVD#:
   $$c_1 = c \cdot 16^{q-p_2}$$
   $$d_1 = \begin{cases} d \cdot 16^{q-p_2} & \text{if CQMPY\#} \\ -d \cdot 16^{q-p_2} & \text{if CQDVD\#.} \end{cases}$$
   Here if the exponent adjustment results in underflow, replace the affected quantity with 0.
4. Compute $u_1 + v_1 i = (ac_1 - bd_1) + (ad_1 + bc_1)i$
5. If CQMPY#, restore the scaling to obtain the answer $u + vi$:
   $u = u_1 \cdot 16^{p_2-q}$ and $v = v_1 \cdot 16^{p_2-q}$.

6. If CQDVD#, compute the denominator as follows:

   $$w_1 = (c_1^2 + d_1^2) \cdot 16^{-2q}$$

   Note that $16^{-2} \leqq w_1 < 2$.

   Then divide: $u_2 = u_1/w_1$ and $v_2 = v_1/w_1$

   Finally, restore the scaling to obtain the answer $u + vi$:

   $$u = u_2 \cdot 16^{-q-p_2} \text{ and } v = v_2 \cdot 16^{-q-p_2}.$$

Effect of an Argument Error

In terms of complex vector relative errors, $\epsilon \sim \delta x + \delta y$ where δx is the relative error of the first operand and δy is the relative error of the second operand.

Complex Exponentiation

(Exponentiation of a Complex Base to an Integer Power)

FCDXI# (COMPLEX*16 Arguments)

FCXPI# (COMPLEX*8 Arguments)

Algorithm

The value of $y_1 + y_2 i = (z_1 + z_2 i)^j$ is computed as follows.

Let $|j| = \sum_{k=0}^{K} r_k \cdot 2^k$ where $r_k = 0$ or 1 for $k = 0, 1, \ldots, K$.

Then $z|j| = \prod_{r_k \neq 0} z^{2^k}$, and the factors z^{2^k}, can be obtained by successive squaring.

More specifically:

1. Initially: $k = 0$, $n^{(0)} = |j|$, $y_1^{(0)} + y_2^{(0)} i = 1 + 0i$, $z_1^{(0)} + z_2^{(0)} i = z_1 + z_2 i$.
2. Raise the index k by 1, and let $n^{(k-1)} = 2q + r$, where q is the integer quotient and $r = 0$ or 1.
3. Let $n^{(k)} = q$.
4. If $r = 0$, then $y_1^{(k)} + y_2^{(k)} i = y_1^{(k-1)} + y_2^{(k-1)} i$.
   If $r = 1$, then $y_1^{(k)} + y_2^{(k)} i = (y_1^{(k-1)} + y_2^{(k-1)} i)\ (z_1^{(k-1)} + z_2^{(k-1)} i)$.
5. If $n^{(k)} \neq 0$, then $z_1^{(k)} + z_2^{(k)} i = (z_1^{(k-1)} + z_2^{(k-1)})^2$, and steps 2 through 5 are repeated until $n^{(k)} = 0$.
6. When $n^{(k)} = 0$, and $j \geqq 0$, then $y_1 + y_2 i = y_1^{(k)} + y_2^{(k)} i$.
   If $j < 0$, then $y_1 + y_2 i = (1 + 0i)\ /\ (y_1^{(k)} + y_2^{(k)} i)$.

(Exponentiation of a Complex Base to a Complex Power)

FCQCQ#(COMPLEX*32 Arguments)
FCDCD# (COMPLEX*16 Arguments)
FCXPC# (COMPLEX*8 Arguments)

Algorithm

$z_1 ** z_2 = \exp(z_2 * \log z_1)$, where the functions 'exp' and 'log' are CEXP and CLOG, CDEXP and CDLOG, or CQEXP and CQLOG respectively as the arguments are C*8, C*16, or C*32.

Effect of an Argument Error

If $z_1 = x_1 + iy_1$, and $z_2 = x_2 + iy_2$, then
$z_1 ** z_2 = \exp(a) * (\cos(b) + i \sin(b))$, where
$a = x_2 * \log|x_1 + iy_1| - y_2 * \arctan(y_1/x_1)$ and
$b = y_2 * \log|x_2 + iy_2| + x_2 * \arctan(y_1/x_1)$.

The function $z ** z_2$ 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.

FCQXI# (COMPLEX*32 Arguments)

Algorithm

1. Write $(x + yi)^J = a + bi$.
2. If $x + yi = 0 + 0i$ and $J > 0$, then $a + bi = 0 + 0i$
3. If $J = 0$, $a + bi = 1.0 + 0i$. Assume now $J \neq 0$.
4. Let $|J| = \sum_{j=0}^{n} g_j 2^{n-j}$ where $g_j = 0$ or 1, $g_0 = 1$.

   Initialize $a_0 + b_0 i = x + yi$. If $|J| = 1$, skip the following.

   Do the following for $j = 1, 2, \ldots, n$:

   $$a_j + b_j i = \begin{cases} (a_{j-1} + b_{j-1}i)^2 & \text{if } g_j = 0 \\ (a_{j-1} + b_{j-1}i)^2(x + yi) & \text{if } g_j = 1 \end{cases}$$

   At the end of iteration $a_n + b_n i = (x + yi)^{|J|}$.
5. If $J < 0$, $(x + yi)^J = \dfrac{1.0 + 0i}{(x + yi)^{|J|}}$

Effect of an Argument Error

$|\epsilon| \sim J\,|\delta|$ where δ is the complex relative error of the base and ϵ is the complex relative error of the result due to this.

Exponentiation of a Real Base to a Real Power

FDXPD# (REAL*8 Arguments)

FRXPR# (REAL*4 Arguments)

Algorithm

Assume the desired answer is a^b.

1. If $a = 0$ and $b \leqq 0$, error return.
   If $a = 0$ and $b > 0$, the answer is 0.
2. If $a \neq 0$ and $b = 0$, the answer is 1.
3. All other cases, compute a^b as $e^{b \cdot \log a}$. In this computation the exponential subroutine and the natural logarithm subroutine are used. If a is negative or if $b \cdot \log a$ is too large, an error return is given by one of these subroutines.

Error estimate

The relative error of the answer can be expressed as $(\epsilon_1 + \epsilon_2)\, b \cdot \log(a) + \epsilon_3$ where ϵ_1, ϵ_2, and ϵ_3 are relative errors of the logarithmic routine, machine multiplication, and the exponential routine, respectively.

For FDXPD#, $\epsilon_1 \leqq 3.5 \times 10^{-16}$, $\epsilon_2 \leqq 2.2 \times 10^{-16}$, and $\epsilon_3 \leqq 2.0 \times 10^{-16}$. Hence the relative error $\leqq 5.7 \times 10^{-16} \times |b \cdot \log a| + 2.0 \times 10^{-16}$. Note that $b \cdot \log a$ is the natural logarithm of the answer.

For FRXPR#, $\epsilon_1 \leqq 8.3 \times 10^{-7}$, $\epsilon_2 \leqq 9.5 \times 10^{-7}$, and $\epsilon_3 \leqq 4.7 \times 10^{-7}$. Hence the relative error $\leqq 1.8 \times 10^{-6} \times |b \cdot \log a| + 4.7 \times 10^{-7}$.

Effect of an Argument Error

$[a(1 + \delta_1)]\, b(1 + \delta_2) \cong a^b(1 + \delta_2 b \cdot \log a + b\delta_1)$. Note that if the answer does not overflow, $|b \cdot \log a| < 175$. On the other hand b can be very large without causing an overflow of a^b if $\log a$ is very small. Thus, if $a \cong 1$ and if b is very large, then the effect of the perturbation δ_1 of a shows very heavily in the relative error of the answer.

FQXPQ# (REAL*16 Arguments)

Algorithm

1. Basically, $x^y = 2^{y \cdot \log_2(x)}$.
2. More specifically, $\log_2(x)$ is computed with aimed accuracy of 16^{-30} in absolute error, or 16^{-28} in relative error, whichever is smaller, by the algorithm of QLOG/QLOG10. The result is kept as two components; the high order part is represented by a single precision number; and the low order part, which is less than 16^{-2} in absolute value, is represented by an extended precision number.
3. The product $y \cdot \log_2(x)$ is obtained by a simulated multiplication to obtain up to 31 hexadecimal digits of accuracy.
4. Raise the result to the power of 2 by the algorithm of QEXP. As stated there, this includes a virtual final rounding with the result that one obtains the full 28 hexadecimal digit accuracy unless x is very close to 1.0.

Effect of an Argument Error

$\epsilon \sim y \cdot \delta_x + y \cdot \log(x) \cdot \delta_y$. The factor $y \cdot \log(x)$ is limited by 180 in magnitude. If beyond this, the result will overflow. However, the other factor y can be very large if x is close to 1. This is so because $\log(x)$ will then be very small.

Exponentiation of a Real Base to an Integer Power

FDXPI# (REAL*8 Arguments)

FRXPI# (REAL*4 Arguments)

Algorithm

Assume the desired answer is a^j.

1. If $a = 0$ and $j \leqq 0$, error return.
   If $a = 0$ and $j > 0$, the answer is 0.
2. If $a \neq 0$ and $j = 0$, the answer is 1.
3. The value of $y = a^j$ is computed as follows: Let $|j| = \sum_{k=0}^{K} r_k 2^k$ where $r_k = 0$ for $k = 0, 1, \ldots, K$. Then $a^{|j|} = \prod_{r_k \neq 0} a^{2^k}$ and the factors a^{2^k} can be obtained by successive squaring.

More specifically:

1. Initially: $k = 0$, $n^{(0)} = |j|$, $y^{(0)} = 1$, and $z^{(0)} = a$.
2. Raise the index k by 1, and decompose $n^{(k-1)} = 2q + r$, where q is the integer quotient and $r = 0$ or 1.
3. Let $n^{(k)} = q$.
4. If $r = 0$, then $y^{(k)} = y^{(k-1)}$.
   If $r = 1$, then $y^{(k)} = y^{(k-1)} z^{(k-1)}$.
5. If $n^{(k)} \neq 0$, then $z^{(k)} = z^{(k-1)} z^{(k-1)}$, and steps 2 through 5 are repeated until $n^{(k)} = 0$.
6. When $n^{(k)} = 0$, and $j \geqq 0$, then $y = y^{(k)}$. If $j < 0$, then $y = \frac{1}{y^{(k)}}$.

Note: The negative exponent is computed by taking the reciprocal of the positive power. Thus it is not possible to compute 16^{-64} because there is a lack of symmetry for real floating-point numbers – i.e., 16^{-64} can be represented, but 16^{64} cannot. The result is obtained by successive multiplications and is exact only if the answer contains at most 14 significant hexadecimal digits.

FQXPI# (REAL*16 Arguments)

Algorithm

1. Write $x^J = y$
2. If $x = 0$ and $J > 0$, then $y = 0$
3. If $x \neq 0$, and $J = 0$, then $y = 1.0$. Assume now $J \neq 0$.
4. Let $|J| = \sum_{j=0}^{n} g_j \, 2^{n-j}$ where $g_j = 0$ or 1, $g_0 = 1$.

Initialize $y_0 = x$. If $|J| = 1$, skip the following.

Do the following for $j = 1, 2, \ldots, n$:

$$y_j = \begin{cases} y^2_{j-1} & \text{if } g_j = 0 \\ y^2_{j-1} \cdot x & \text{if } g_j = 1 \end{cases}$$

At the end of iteration $y_n = x^{|J|}$.

5. If $J < 0, x^J = \dfrac{1}{x^{|J|}}$

Note: The negative power is computed by taking the reciprocal of the positive power. Thus it is not possible to compute 16^{-64} because there is a lack of symmetry in real floating point numbers; i.e., 16^{-64} can be represented, but 16^{64} cannot.

Effect of an Argument Error

$\epsilon \sim J\,\delta$

Exponentiation of an Integer Base to an Integer Power

FIXPI# (INTEGER*4 Arguments)

Algorithm

Assume the desired answer is I^j.

1. If $I = 0$ and $j \leqq 0$, error return.
   If $I = 0$ and $j > 0$, the answer is 0.
2. If $I \neq 0$ and $j = 0$, the answer is 1.
3. The value of $L = I^j$ is computed as follows: Let $j = \sum_{k=0}^{K_1} r_k \cdot 2^k$ where $r_k = 0$ or 1 for $k = 0, 1, \ldots, K$. Then $I^j = \prod_{r_k \neq 0} I^{2^k}$, and the factors I^{2^k} can be obtained by successive squaring.

   More specifically:
   1. Initially: $k = 0$, $n^{(0)} = j$, $y^{(0)} = 1$, and $m^{(0)} = I$.
   2. Raise the index k by 1, and decompose $n^{(k-1)} = 2q + r$, where q is the integer quotient and $r = 0$ or 1.
   3. Let $n^{(k)} = q$.
   4. If $r = 0$, then $y^{(k)} = y^{(k-1)}$.
      If $r = 1$, then $y^{(k)} = y^{(k-1)} \cdot m^{(k-1)}$.
   5. If $n^{(k)} \neq 0$, then $m^{(k)} = m^{(k-1)} \cdot m^{(k-1)}$, and steps 2 through 5 are repeated until $n^{(k)} = 0$.
   6. When $n^{(k)} = 0$, $L = L^{(k)}$.

   Note: The result is obtained by successive multiplications. The result is exact only if it is less than $2^{31} - 1$. Results are meaningless when this limit is exceeded and may even be of changed sign. No tests for overflow are made.

Exponentiation of a Base 2 Argument to a Real Power

FQXP2# (REAL*16 Arguments)

Algorithm

This subprogram uses the same algorithm as the QEXP explicit subprogram.

Appendix E. Storage Estimates

This appendix contains decimal storage estimates (in bytes) for the library subprograms. The estimate given does not include any additional mathematical subprograms for VS FORTRAN routines that the subprograms may use during execution. The entry-names of any additional mathematical library subprograms used are shown in Figure 43. Figure 43 through Figure 46 on page 422, also indicate which mathematical, service, character, and bit subprograms require VS FORTRAN routines for input, output, interruption, and error procedures. The table on page 354 shows storage estimates for library execution-time routines.

The programmer must add the estimates for all subprograms and routines needed to determine the amount of storage required. If the programmer has not made allowances for the storage required by any of these additional routines, the amount of available storage may be exceeded and execution cannot begin (or may terminate abnormally).

Entry Name(s)	Decimal Estimate (in Bytes)	Subprogram Used	I/O, Error, and Interrupt Routines
ABS,DABS,IABS,QABS	216		No
ACOS,ASIN	572	SQRT	Yes
AIMAG,DIMAG,QIMAG	184		No
AINT,DINT,QINT	344		No
ALGAMA,GAMMA,LGAMMA	736	ALOG,EXP	Yes
ALOG,LOG	696		Yes
ALOG10,LOG10	716		Yes
AMOD,DMOD,QMOD	276		No
ANINT,DNINT,NINT,IDNINT	504		No
ATAN,ATAN2	648		Yes
CABS	336	SQRT	Yes
CCOS,CSIN	716	EXP,SIN/COS	Yes
CDABS	348	DSQRT	Yes
CDCOS,CDSIN	752	DEXP,DSIN/DCOS	Yes

Figure 43 (Part 1 of 4). Mathematical Subprogram Storage Estimates

Entry Name(s)	Decimal Estimate (in Bytes)	Subprogram Used	I/O, Error, and Interrupt Routines
CDDVD#	216		No
CDEXP	552	DEXP,DSIN/DCOS	Yes
CDLOG	576	DLOG,ATAN2	Yes
CDMPY#	148		No
CDSQRT	476	DSQRT	Yes
CDVD#	212		No
CEXP	532	EXP,SIN/COS	Yes
CLOG	556	ALOG,ATAN2	Yes
CMPY#	148		No
CONJG,DCONJG,QCONJG	208		No
COS	700		Yes
COSH,SINH	560	EXP	Yes
COTAN,TAN	652		Yes
CQABS	360	QSQRT	No [3]
CQCOS,CQSIN	1140	QEXP,QSIN,QCOS	Yes
CQDVD#,CQMPY#	812		No
CQEXP	600	QEXP,QSIN,QCOS	Yes
CQLOG	696	QLOG,QATAN2	Yes
CQSQRT	520	QSQRT	No [3]
CSQRT	456	SQRT	Yes
DACOS,DASIN	680	DSQRT	Yes
DATAN,DATAN2	788		Yes
DCOS	748		Yes
DCOS[1]	800		Yes
DCOSH,DSINH	648	DEXP	Yes
DCOTAN,DTAN	764		Yes
DCOTAN,DTAN[1]	904		Yes
DDIM,DIM,IDIM,QDIM	264		No
DERF,DERFC	992	DEXP	Yes
DEXP	1096		Yes
DEXP[1]	780		Yes
DGAMMA,DLGAMA	948	DLOG,DEXP	Yes
DLOG	832		Yes

Figure 43 (Part 2 of 4). Mathematical Subprogram Storage Estimates

Entry Name(s)	Decimal Estimate (in Bytes)	Subprogram Used	I/O, Error, and Interrupt Routines
DLOG10	840		Yes
DPROD	220		No
DSIGN,ISIGN,SIGN,QSIGN	272		No
DSIN	860		Yes
DSIN[1]	800		Yes
DSQRT	476		Yes
DTANH	456	DEXP	Yes
EXP	696		Yes
EXP[1]	560		Yes
ERF,ERFC	708	EXP	Yes
FCDCD#	656	CDMPY#,CDLOG,CDEXP	Yes
FCDXI#	612	CDMPY#/CDDVD#	Yes
FCQCQ#	760	CQEXP,CQMPY#,CQDVD#, CQLOG	Yes
FCQXI#	648	CQMPY#,CQDVD#	Yes
FCXPC#	612	CMPY#,CLOG,CEXP	Yes
FCXPI#	576	CDVD#/CMPY#	Yes
FDXPD#	1704	DEXP,DLOG	Yes
FDXPD#[1]	1712		Yes
FDXPI#	492		Yes
FIXPI#	464		Yes
FQXPI#	516		Yes
FQXPQ#,FQXP2#	2560		Yes
FRXPI#	480		No
FRXPR#	1228	EXP,ALOG	Yes
FRXPR#[1]	1380		Yes
MOD	132		No
QARCOS,QARSIN	1196	QSQRT	Yes
QATAN,QATAN2	1392		Yes
QCOS,QSIN	1104		Yes
QCOSH,QSINH	952	QEXP	Yes
QCOTAN,QTAN	1208		Yes
QERF,QERFC[2]	1648	QEXP	Yes
QEXP,QLOG,QLOG10	2560		Yes

Figure 43 (Part 3 of 4). Mathematical Subprogram Storage Estimates

Entry Name(s)	Decimal Estimate (in Bytes)	Subprogram Used	I/O, Error, and Interrupt Routines
QSQRT	648		Yes
QTANH	772	QEXP	No[3]
SIN	708		Yes
SQRT	488		Yes
TANH	388	EXP	Yes

Figure 43 (Part 4 of 4). Mathematical Subprogram Storage Estimates

Notes to Figure 43 :

1 This entry name is an alternative mathematical library subroutine name.

2 When the argument falls between 2.84375 and 13.306, the module IFYQERF2 (size 1300 bytes) is also used. IFYQERF2, in turn, uses routine FQXPQ#.

3 Although this mathematical subprogram does not itself require the input, output, error, or interruption routines, it does use other mathematical subprograms that do.

Entry Name(s)	Decimal Estimate (in bytes)	I/O, Error, and Interrupt Routines
CDUMP/CPDUMP	270	Yes
DSPAN#,DSPN2#	186	Yes
DSPN4#	140	Yes
DUMP/PDUMP	270	Yes
DVCHK	308	Yes
DYCMN#	114	Yes
EXIT	188	Yes
OVERFL	324	Yes
SDUMP	114	Yes
XUFLOW	272	No

Figure 44. Service Subprogram Storage Estimates

Entry Name(s)[1]	Decimal Estimate (in bytes)	I/O, Error, and Interrupt Routines
CCMPR#,CXMPR#[2]	596	Yes[3]
CHAR,ICHAR,LEN	464	Yes
CMOVE#	416	Yes
CNCAT#	424	Yes
INDEX	544	Yes
LGE,LGT,LLE,LLT	1916	Yes

Figure 45. Character Subprogram Storage Estimates

Notes to Figure 45 :

1 No additional character subprograms are used.

2 The entry point CXMPR# is used for complex operands.

3 There is no I/O error or interrupt routine invoked for the CXMPR# entry name.

Entry Name(s)	Decimal Estimate (in bytes)	I/O, Error, and Interrupt Routines
BTEST,IBCLR,IBSET,ISHFT	696	Yes
IAND,IEOR,IOR,NOT	82	No

Figure 46. Bit Subprogram Storage Estimates

Routine Name	Decimal Estimate (Bytes)
IFYCLBC0[3]	4784
IFYCREN[3]	252
IFYCVIO$[3]	136
IFYCVIOS[3]	7172
IFYDBDFT	24
IFYDCOM2[4]	1920
IFYDDCMN[4]	114
IFYDDCMP[4]	624
IFYDDIOS[4]	4784
IFYDFNTH[4]	2276
IFYDIOCS	118
IFYDLBC0[4]	4784
IFYDKIOS[4, 5]	12688
IFYDREN[4]	246
IFYDSIOS[4]	7762
IFYDSPAN	186
IFYDSPAP[4]	1308
IFYDVIOS[4]	7836
IFYIBCOM	1472
IFYIBCOP	1956
IFYLDFIO	482
IFYLDFIP	1056
IFYNAMEL	282
IFYNAMEP	900
IFYOPSYP[4]	460
IFYOPSYS[4]	118
IFYSDUMQ	8534
IFYTFORT	192
IFYUATBL[1, 2]	---
IFYUOPT[2]	1464
IFYVAREN	216
IFYVASUB	3356
IFYVASYN[5]	220
IFYVASYP	2752
IFYVBLN$	52
IFYVBLNT	492
IFYVBREN	252
IFYVCIAD	2950
IFYVCIA4	3402
IFYVCLMI	312
IFYVCLOP	560
IFYVCLOS	114
IFYVCLSI	752
IFYVCMSS[3]	944
IFYVCNI$	120
IFYVCNO$	120
IFYVCOM[2]	3020
IFYVCOM$	52
IFYVCOMH	5378
IFYVCONI	732
IFYVCONO	1876
IFYVCVT$	780
IFYVCVTH	4444
IFYVDBUG	708
IFYVDBUP	58
IFYVDIO$	58
IFYVDIOS[5]	5956
IFYVDUMQ	1748
IFYVEMG$	116
IFYVERE$	52
IFYVERRE	540
IFYVERRM	2017
IFYVERS$	96
IFYVFNTH	2283
IFYVIIO$	58
IFYVIIOS	636
IFYVINQP	2404
IFYVINQR	114
IFYVINTE	472
IFYVIOCP	662
IFYVIOCT	274
IFYVIOD0	13
IFYVIOFM	858
IFYVIOFP	1704
IFYVIOK0	13
IFYVIOLD	524
IFYVIOLP	4584
IFYVIONL	378
IFYVIONP	4134
IFYVIOUF	666
IFYVIOUP	2744
IFYVKIO$	58
IFYVKIOS[5]	10864
IFYVLBC0	4810
IFYVLCI0	148
IFYVLINK	366
IFYVLINP	564
IFYVLOAD	688
IFYVLOC$	52
IFYVLOCA	1403
IFYVMOPP	1104
IFYVMOPT	308
IFYVOPEN	114
IFYVOPEP	1672
IFYVPARM	684
IFYVPOSA	11694
IFYVPOST	11678
IFYVPOS$	52
IFYVREN	258
IFYVSCOM	1064
IFYVSCOP	1536
IFYVSERH	340
IFYVSFIO	2146
IFYVSFST	7306
IFYVSIOS[5]	8920
IFYVSPAN	140
IFYVSPAP	1132
IFYVSPIE	308
IFYVSTAE	2108
IFYVSTA$	268
IFYVTEN	704
IFYVTRC$	52
IFYVTRCH	2188
IFYVVIOS[5]	6580
IFYVXMSK	272

Figure 47. Table of Storage Estimates for Library Execution-Time Routines

Notes to Figure 47

1 The number of bytes in table IFYUATBL may be computed by the formula $16n + 8$, where n is the number of unit numbers requested during installation.

2 The size of this routine is installation dependent.

3 This routine is for CMS only.

4 This routine is for VSE/Advanced Functions.

5 This module also requires dynamic storage. For each I/O file used, the amount (in bytes) is 256 plus buffer size(s).

Appendix F. Accuracy Statistics

This appendix contains accuracy statistics for explicitly and implicitly called mathematical subprograms. These statistics are presented in Figure 48 on page 427. They are arranged in alphabetic order, according to the entry names. The following information is given:

Entry Name: This column gives the entry name used to call the subprogram.

Argument Range: This column gives the argument range used to obtain the accuracy figures. For each function, accuracy figures are given for one or more representative segments within the valid range. In each case, the figures given are the most meaningful to the function and range under consideration.

The maximum relative error and standard deviation of the relative error are generally useful and revealing statistics; however, they are useless for the range of a function where its value becomes 0. This is because the slightest error in the argument can cause an unpredictable fluctuation in the magnitude of the answer. When a small argument error would have this effect, the maximum absolute error and standard deviation of the absolute error are given for the range.

Sample: This column indicates the type of sample used for the accuracy figures. The type of sample depends on the function and range under consideration. The statistics may be based either upon an exponentially distributed (E) argument sample or a uniformly distributed (U) argument sample.

Accuracy Figures: This column gives accuracy figures for one or more representative segments within the valid argument range. The accuracy figures supplied are based on the assumption that the arguments are perfect, that is, without error and, therefore, have no error propagation effect upon the answers. The only errors in the answer are those introduced by the subprograms. Appendix D, "Algorithms for Library Mathematical Functions" on page 369, contains a description of some of the symbols used in this appendix; the following additional symbols are used in the presentation of accuracy figures:

Formula	Description
$M(\epsilon) = \text{Max}\left\lvert\frac{f(x) - g(x)}{f(x)}\right\rvert$	The maximum relative error produced during testing.
$\sigma(\epsilon) = \sqrt{\frac{1}{N}\sum_i \left\lvert\frac{f(x_i) - g(x_i)}{f(x_i)}\right\rvert^2}$	The standard deviation (root-mean-square) of the relative error.
$M(E) = \text{Max}\lvert f(x) - g(x)\rvert$	The maximum absolute error produced during testing.
$\sigma(E) = \sqrt{\frac{1}{N}\sum_i \lvert f(x_i) - g(x_i)\rvert^2}$	The standard deviation (root-mean-square) of the absolute error.

In case of complex functions, the absolute value signs employed in the above definitions are to mean the complex absolute values. In the formulas for standard deviation, *N* represents the total number of arguments in the sample; *i* is a subscript that varies from 1 to N.

Accuracy statistics for the alternative mathematical library subroutines can be found in the articles listed in the Preface.

Entry Name	Argument Range	Sample E/U	Accuracy Figures			
			Relative		Absolute	
			M (ε)	σ (ε)	M (E)	σ (E)
ALGAMA	$0 < X < 0.5$	U	1.16×10^{-6}	3.54×10^{-7}		
	$0.5 \leqq X < 3.0$	U			9.43×10^{-7}	3.42×10^{-7}
	$3.0 \leqq X < 8.0$	U	1.25×10^{-6}	3.04×10^{-7}		
	$8.0 \leqq X < 16.0$	U	1.18×10^{-6}	3.80×10^{-7}		
	$16.0 \leqq X < 500.0$	U	9.85×10^{-7}	1.90×10^{-7}		
ALOG	$0.5 \leqq X \leqq 1.5$	U			6.85×10^{-8}	2.33×10^{-8}
	$X < 0.5, X > 1.5$	E	8.32×10^{-7}	1.19×10^{-7}		
ALOG10	$0.5 \leqq X \leqq 1.5$	U			7.13×10^{-8}	2.26×10^{-8}
	$X < 0.5, X > 1.5$	E	1.05×10^{-6}	2.17×10^{-7}		
ACOS	$-1 \leqq X \leqq +1$	U	8.85×10^{-7}	3.19×10^{-7}		
ASIN	$-1 \leqq X \leqq +1$	U	9.34×10^{-7}	2.06×10^{-7}		
ATAN	The full range	Note 7	1.01×10^{-6}	4.68×10^{-7}		
ATAN2	The full range	Note 7	1.01×10^{-6}	4.68×10^{-7}		
CABS	The full range	Note 1	9.15×10^{-7}	2.00×10^{-7}		
CCOS	$\|X_1\| \leqq 10, \|X_2\| \leqq 1$	U	2.50×10^{-6} See Note 2	7.66×10^{-7}		
CDABS	The full range	Note 1	2.03×10^{-16}	4.83×10^{-17}		
CDCOS	$\|X_1\| \leqq 10, \|X_2\| \leqq 1$	U	3.98×10^{-15} See Note 3	2.50×10^{-16}		
CDEXP	$\|X_1\| \leqq 1, \|X_2\| \leqq \pi/2$	U	3.76×10^{-16}	1.10×10^{-16}		
	$\|X_1\| \leqq 20, \|X_2\| \leqq 20$	U	2.74×10^{-15}	9.64×10^{-16}		
CDLOG	The full range except (1 + 0i)	Note 1	2.72×10^{-16}	5.38×10^{-17}		
CDSIN	$\|X_1\| \leqq 10, \|X_2\| \leqq 1$	U	2.35×10^{-15} See Note 4	2.25×10^{-16}		
CDSQRT	The full range	Note 1	1.76×10^{-16}	4.06×10^{-17}		
CEXP	$\|X_1\| \leqq 170, \|X_2\| \leqq \pi/2$	U	9.93×10^{-7}	2.67×10^{-7}		
	$\|X_1\| \leqq 170, \pi/2 < \|X_2\| \leqq 20$	U	1.07×10^{-6}	2.73×10^{-7}		
CLOG	The full range except (1 + 0i)	Note 1	7.15×10^{-7}	1.36×10^{-7}		
COS	$0 \leqq X \leqq \pi$	U			1.19×10^{-7}	4.60×10^{-8}
	$-10 \leqq X < 0, \pi < X \leqq 10$	U			1.28×10^{-7}	4.55×10^{-8}
	$10 < \|X\| \leqq 100$	U			1.14×10^{-7}	4.60×10^{-8}
COSH	$-5 \leqq X \leqq +5$	U	1.27×10^{-6}	2.63×10^{-7}		
COTAN	$\|X\| \leqq \pi/4$	U	1.07×10^{-6}	3.58×10^{-7}		
	$\pi/4 < \|X\| \leqq \pi/2$	U	1.40×10^{-6} (Note 5)	2.56×10^{-7}		
	$\pi/2 < \|X\| \leqq 10$	U	1.30×10^{-6} (Note 5)	3.11×10^{-7}		
	$10 < \|X\| \leqq 100$	U	1.49×10^{-6} (Note 5)	3.15×10^{-7}		
NOTES: (See end of figure.)						

Figure 48 (Part 1 of 6). Accuracy Figures

Entry Name	Argument Range	Sample E/U	Accuracy Figures			
			Relative		Absolute	
			M (ε)	σ (ε)	M (E)	σ (E)
CQABS	The full range	Note 9	2.77×10^{-33}	5.45×10^{-34}		
CQCOS	$-10 < x < 10$ $-1 < y < 1$	U	6.87×10^{-33}	2.44×10^{-33}		
CQDVD#	Note 8	Note 8	5.32×10^{-33}	1.42×10^{-33}		
CQEXP	$-170 < x < 170$ $-\frac{\pi}{2} < y < \frac{\pi}{2}$	U	3.82×10^{-33}	8.30×10^{-34}		
CQLOG	The full range	Note 9	4.53×10^{-33}	9.72×10^{-34}		
CQMPY#	Note 8	Note 8	4.52×10^{-33}	1.27×10^{-33}		
CQSIN	$-10 < x < 10$ $-1 < y < 1$	U	7.26×10^{-33}	2.37×10^{-33}		
CQSQRT	The full range	Note 9	3.37×10^{-33}	7.27×10^{-34}		
CSIN	$\lvert X_1 \rvert \leqq 10$, $\lvert X_2 \rvert \leqq 1$	U	1.92×10^{-6} See Note 6	7.38×10^{-7}		
CSQRT	The full range	Note 1	7.00×10^{-7}	1.71×10^{-7}		
DACOS	$\lvert X \rvert \leqq 1$	U	2.07×10^{-16}	7.05×10^{-17}		
DASIN	$\lvert X \rvert \leqq 1$	U	2.04×10^{-16}	5.15×10^{-17}		
DATAN	The full range	Note 7	2.18×10^{-16}	7.04×10^{-17}		
DATAN2	The full range	Note 7	2.18×10^{-16}	7.04×10^{-17}		
DCOS	$0 \leqq X \leqq \pi$	U			1.79×10^{-16}	6.53×10^{-17}
	$-10 \leqq X < 0$, $\pi < X \leqq 10$	U			1.75×10^{-16}	5.93×10^{-17}
	$10 < X \leqq 100$	U			2.64×10^{-15}	1.01×10^{-15}
DCOSH	$\lvert X \rvert \leqq 5$	U	3.63×10^{-16}	9.05×10^{-17}		
DCOTAN	$\lvert X \rvert \leqq \pi/4$	U	2.46×10^{-16} (Note 5)	8.79×10^{-17}		
	$\pi/4 < \lvert X \rvert \leqq \pi/2$	U	2.78×10^{-13} (Note 5)	8.61×10^{-15}		
	$\pi/2 < \lvert X \rvert \leqq 10$	U	5.40×10^{-13} (Note 5)	1.13×10^{-14}		
	$10 < \lvert X \rvert \leqq 100$	U	8.61×10^{-13} (Note 5)	4.61×10^{-14}		
DERF	$\lvert X \rvert \leqq 1.0$	U	1.89×10^{-16}	2.60×10^{-17}		
	$1.0 < \lvert X \rvert \leqq 2.04$	U	2.87×10^{-17}	9.84×10^{-18}		
	$2.04 < \lvert X \rvert < 6.092$	U	1.39×10^{-17}	8.02×10^{-18}		
DERFC	$-6 < X < 0$	U	2.08×10^{-16}	6.52×10^{-17}		
	$0 \leqq X \leqq 1$	U	1.40×10^{-16}	2.59×10^{-17}		
	$1 < X \leqq 2.04$	U	4.11×10^{-16}	8.86×10^{-17}		
	$2.04 < X < 4$	U	3.26×10^{-16}	8.65×10^{-17}		
	$4 \leqq X < 13.3$	U	3.51×10^{-15}	1.96×10^{-15}		

NOTES: (See end of figure.)

Figure 48 (Part 2 of 6). Accuracy Figures

Entry Name	Argument Range	Sample E/U	Accuracy Figures			
			Relative		Absolute	
			M (ϵ)	σ (ϵ)	M (E)	σ (E)
DEXP	$\|X\| \leqq 1$	U	2.04×10^{-16}	5.43×10^{-17}		
	$1 < \|X\| \leqq 20$	U	2.03×10^{-16}	4.87×10^{-17}		
	$20 < \|X\| \leqq 170$	U	1.97×10^{-16}	4.98×10^{-17}		
DGAMMA	$0 < X < 1$	U	2.14×10^{-16}	7.84×10^{-17}		
	$1 \leqq X \leqq 2$	U	2.52×10^{-17}	6.07×10^{-18}		
	$2 < X < 4$	U	2.21×10^{-16}	8.49×10^{-17}		
	$4 \leqq X < 8$	U	5.05×10^{-16}	1.90×10^{-16}		
	$8 \leqq X < 16$	U	6.02×10^{-15}	1.78×10^{-15}		
	$16 \leqq X < 57$	U	1.16×10^{-14}	4.11×10^{-15}		
DLGAMA	$0 < X \leqq 0.5$	U	2.77×10^{-16}	9.75×10^{-17}		
	$0.5 < X < 3$	U			2.24×10^{-16}	7.77×10^{-17}
	$3 \leqq X < 8$	U	2.89×10^{-16}	8.80×10^{-17}		
	$8 \leqq X < 16$	U	2.86×10^{-16}	8.92×10^{-17}		
	$16 \leqq X < 500$	U	1.99×10^{-16}	3.93×10^{-17}		
DLOG	$0.5 \leqq X \leqq 1.5$	U			4.60×10^{-17}	2.09×10^{-17}
	$X < 0.5, X > 1.5$	E	3.32×10^{-16}	5.52×10^{-17}		
DLOG10	$0.5 \leqq X \leqq 1.5$	U			2.73×10^{-17}	1.07×10^{-17}
	$X < 0.5, X > 1.5$	E	3.02×10^{-16}	6.65×10^{-17}		
DSIN	$\|X\| \leqq \pi/2$	U	3.60×10^{-16}	4.82×10^{-17}	7.74×10^{-17}	1.98×10^{-17}
	$\pi/2 < \|X\| \leqq 10$	U			1.64×10^{-16}	6.49×10^{-17}
	$10 < \|X\| \leqq 100$	U			2.68×10^{-15}	1.03×10^{-15}
DSINH	$\|X\| \leqq 0.88137$	U	2.06×10^{-16}	3.74×10^{-17}		
	$0.88137 < \|X\| \leqq 5$	U	3.80×10^{-16}	9.21×10^{-17}		
DSQRT	The full range	E	1.06×10^{-16}	2.16×10^{-17}		
DTAN	$\|X\| \leqq \pi/4$	U	3.41×10^{-16}	6.27×10^{-17}		
	$\pi/4 < \|X\| \leqq \pi/2$	U	1.43×10^{-12} (Note 5)	2.95×10^{-14}		
	$\pi/2 < \|X\| \leqq 10$	U	2.78×10^{-13} (Note 5)	7.23×10^{-15}		
	$10 < \|X\| \leqq 100$	U	3.79×10^{-12} (Note 5)	9.50×10^{-14}		
DTANH	$\|X\| \leqq 0.54931$	U	1.91×10^{-16}	3.86×10^{-17}		
	$0.54931 < \|X\| \leqq 5$	U	1.54×10^{-16}	1.87×10^{-17}		
ERF	$\|X\| \leqq 1.0$	U	8.16×10^{-7}	1.10×10^{-7}		
	$1.0 < \|X\| \leqq 2.04$	U	1.13×10^{-7}	3.70×10^{-8}		
	$2.04 < \|X\| \leqq 3.9192$	U	5.95×10^{-8}	3.41×10^{-8}		
NOTES: (See end of figure.)						

Figure 48 (Part 3 of 6). Accuracy Figures

Entry Name	Argument Range	Sample E/U	Accuracy Figures			
			Relative		Absolute	
			$M(\epsilon)$	$\sigma(\epsilon)$	$M(E)$	$\sigma(E)$
ERFC	$-3.8 < X < 0$	U	9.10×10^{-7}	2.96×10^{-7}		
	$0 \leqq X \leqq 1.0$	U	7.42×10^{-7}	1.27×10^{-7}		
	$1.0 < X \leqq 2.04$	U	1.54×10^{-6}	3.78×10^{-7}		
	$2.04 < X \leqq 4.0$	U	2.28×10^{-6}	3.70×10^{-7}		
	$4.0 < X \leqq 13.3$	U	1.55×10^{-5}	8.57×10^{-6}		
EXP	$\lvert X\rvert \leqq 1$	U	4.65×10^{-7}	1.28×10^{-7}		
	$1 < \lvert X\rvert \leqq 170$	U	4.42×10^{-7}	1.15×10^{-7}		
FCQXI# Note 11	$2 \leqq J \leqq 160$, $10^{-70/J} < \lvert x + iy\rvert < 10^{75/J}$		$3.7 \times 10^{-33} \times J$	$10^{-33} \times (J - 1)$		
FQXPI#	$2 \leqq J \leqq 160$, $10^{-75/J} < x < 10^{75/J}$		$2.5 \times 10^{-33} \times (J - 1)$	$6.1 \times 10^{-34} \times (J - 1)$		
FQXPQ#	$0.99 < A < 1.01$ $-75 \log_A 10 < 8$ $< 75 \log_A 10$	U	5.68×10^{-31}	5.16×10^{-32}		
FQXP2#	$-260 < x < 252$	U	1.52×10^{-33}	3.78×10^{-34}		
GAMMA	$0 < X < 1.0$	U	9.86×10^{-7}	3.66×10^{-7}		
	$1.0 \leqq X \leqq 2.0$	U	1.13×10^{-7}	3.22×10^{-8}		
	$2.0 < X \leqq 4.0$	U	9.47×10^{-7}	3.79×10^{-7}		
	$4.0 < X < 8.0$	U	2.26×10^{-6}	8.32×10^{-7}		
	$8.0 \leqq X \leqq 16.0$	U	2.20×10^{-5}	7.61×10^{-6}		
	$16.0 < X \leqq 57.0$	U	4.62×10^{-5}	1.51×10^{-5}		
QARCOS	$-1 \leqq x \leqq 1$	U	3.18×10^{-33}	9.81×10^{-34}		
QARSIN	$-1 \leqq x \leqq 1$	U	3.14×10^{-33}	7.89×10^{-34}		
QATAN	$-10^{75} < x < 10^{75}$	Note 10	2.92×10^{-33}	7.32×10^{-34}		
QATAN2	The full range	Note 9	3.53×10^{-33}	7.83×10^{-34}		
QCOS	$0 \leqq x \leqq \pi$	U	4.41×10^{-33}	6.58×10^{-34}	3.23×10^{-34}	1.48×10^{-34}
	$-10 < x < 0$, or $\pi \leqq x < 10$	U			3.43×10^{-34}	1.57×10^{-34}
	$-200 < x \leqq -10$, or $10 \leqq x < 200$	U			3.48×10^{-34}	1.57×10^{-34}
QCOSH	$-10 < x < 10$	U	5.83×10^{-33}	1.57×10^{-33}		

NOTES: (See end of figure.)

Figure 48 (Part 4 of 6). Accuracy Figures

Entry Name	Argument Range	Sample E/U	Accuracy Figures			
			Relative		Absolute	
			$M(\epsilon)$	$\sigma(\epsilon)$	$M(E)$	$\sigma(E)$
QCOTAN	$-\frac{\pi}{4} < x < \frac{\pi}{4}$	U	3.02×10^{-33}	9.09×10^{-34}		
	$-\frac{\pi}{2} < x \leqq -\frac{\pi}{4}$, or $\frac{\pi}{4} \leqq x < \frac{\pi}{2}$	U	3.98×10^{-33}	1.09×10^{-33}		
	$-10 < x \leqq -\frac{\pi}{2}$, or $\frac{\pi}{2} \leqq x < 10$	U	4.55×10^{-33}	1.13×10^{-33}		
	$-200 < x \leqq -10$, or $10 \leqq x < 200$	U	3.98×10^{-33}	1.11×10^{-33}		
QERF	$\lvert x \rvert < 1$	U	3.0×10^{-33}	5.3×10^{-34}		
	$1 \leqq \lvert x \rvert < 2.84375$	U	9.2×10^{-34}	2.3×10^{-34}		
	$2.84375 \leqq \lvert x \rvert < 5$	U	1.9×10^{-34}	1.3×10^{-34}		
QERFC	$-5 < x < 0$	U	3.1×10^{-33}	1.2×10^{-33}		
	$0 \leqq x < 1$	U	3.3×10^{-33}	5.8×10^{-34}		
	$1 \leqq x < 2.84375$	U	7.7×10^{-33}	2.8×10^{-33}		
	$2.84375 \leqq x < 5$	U	4.88×10^{-32}	1.83×10^{-32}		
QEXP	$-1 < x < 1$	U	1.51×10^{-33}	4.27×10^{-34}		
	$-10 < x < 10$	U	1.53×10^{-33}	3.96×10^{-34}		
	$-180 < x < 174$	U	1.54×10^{-33}	3.82×10^{-34}		
QLOG	$0.99 < x < 1.01$	U	4.27×10^{-33}	1.51×10^{-33}	1.92×10^{-35}	8.36×10^{-36}
	$0.5 < x < 2$	U	4.06×10^{-33}	8.24×10^{-34}	3.17×10^{-34}	1.63×10^{-34}
	$10^{-78} < x < 10^{75}$	E	4.45×10^{-33}	8.77×10^{-34}		
QLOG10	$10^{-78} < x < 10^{75}$	E	3.59×10^{-33}	1.16×10^{-33}		
QSIN	$-\frac{\pi}{2} < x < \frac{\pi}{2}$	U	2.48×10^{-33}	3.12×10^{-34}	2.95×10^{-34}	1.17×10^{-34}
	$-10 < x \leqq -\frac{\pi}{2}$, or $\frac{\pi}{2} \leqq x < 10$	U			3.48×10^{-34}	1.60×10^{-34}
	$-200 < x \leqq -10$, or $10 \leqq x < 200$	U			3.50×10^{-34}	1.56×10^{-34}
QSINH	$-1 < x < 1$	U	2.91×10^{-33}	6.86×10^{-34}		
	$-10 < x \leqq -1$, or $1 \leqq x < 10$	U	6.71×10^{-33}	1.37×10^{-33}		
QSQRT	$10^{-50} < x < 10^{50}$	E	1.49×10^{-33}	2.95×10^{-34}		
	$10^{-78} < x < 10^{75}$	E	1.39×10^{-33}	2.76×10^{-34}		
QTAN	$-\frac{\pi}{4} < x < \frac{\pi}{4}$	U	3.75×10^{-33}	9.16×10^{-34}		
	$-\frac{\pi}{2} < x \leqq \frac{\pi}{4}$, or $\frac{\pi}{4} \leqq x < \frac{\pi}{2}$	U	2.77×10^{-33}	8.78×10^{-34}		
	$-10 < x \leqq -\frac{\pi}{2}$, or $\frac{\pi}{2} \leqq x < 10$	U	4.52×10^{-33}	9.16×10^{-33}		
	$-200 < x \leqq -10$, or $10 \leqq x < 200$	U	4.47×10^{-33}	9.12×10^{-33}		
QTANH	$-0.54931 < x < 0.54931$	U	2.41×10^{-33}	5.12×10^{-34}		
	$-5 < x \leqq -0.54931$, or $0.54931 \leqq x < 5$	U	2.09×10^{-33}	2.46×10^{-34}	1.04×10^{-33}	1.68×10^{-34}

NOTES: (See end of figure.)

Figure 48 (Part 5 of 6). Accuracy Figures

Entry Name	Argument Range	Sample E/U	Accuracy Figures			
			Relative		Absolute	
			M (ε)	σ (ε)	M (E)	σ (E)
SIN	$\|X\| \leqq \pi/2$	U	1.32×10^{-6}	1.82×10^{-7}	1.18×10^{-7}	4.55×10^{-8}
	$\pi/2 < \|X\| \leqq 10$	U			1.15×10^{-7}	4.64×10^{-8}
	$10 < \|X\| \leqq 100$	U			1.28×10^{-7}	4.52×10^{-8}
SINH	$-5 \leqq X \leqq +5$	U	1.26×10^{-6}	2.17×10^{-7}		
SQRT	The full range	E	4.45×10^{-7}	8.43×10^{-8}		
TAN	$\|X\| \leqq \pi/4$	U	1.71×10^{-6}	2.64×10^{-7}		
	$\pi/4 < \|X\| \leqq \pi/2$	U	1.05×10^{-6} (Note 5)	3.59×10^{-7}		
	$\pi/2 < \|X\| \leqq 10$	U	6.49×10^{-6} (Note 5)	3.38×10^{-7}		
	$10 < \|X\| \leqq 100$	U	1.57×10^{-6} (Note 5)	3.07×10^{-7}		
TANH	$\|X\| \leqq 0.7$	U	8.48×10^{-7}	1.48×10^{-7}		
	$0.7 < \|X\| \leqq 5$	U	2.44×10^{-7}	4.23×10^{-8}		

NOTES:

[1] The distribution of sample arguments upon which these statistics are based is exponential radially and is uniform around the origin.

[2] The maximum relative error cited for the CCOS function is based upon a set of 2000 random arguments within the range. In the immediate proximity of the points $\left(n + \frac{1}{2}\right)\pi + 0i$ (where $n = 0, \pm 1, \pm 2, \ldots,$) the relative error can be quite high, although the absolute error is small.

[3] The maximum relative error cited for the CDCOS function is based upon a set of 1500 random arguments within the range. In the immediate proximity of the points $\left(n + \frac{1}{2}\right)\pi + 0i$ (where $n = 0, \pm 1, \pm 2, \ldots,$) the relative error can be quite high, although the absolute error is small.

[4] The maximum relative error cited for the CDSIN function is based upon a set of 1500 random arguments within the range. In the immediate proximity of the points $n\pi + 0i$ (where $n = \pm 1, \pm 2, \ldots,$) the relative error can be quite high, although the absolute error is small.

[5] The figures cited as the maximum relative errors are those encountered in a sample of 2500 random arguments within the respective ranges. See Appendix D, "Algorithms for Library Mathematical Functions" for a description of the behavior of errors when the argument is near a singularity or a zero of the function.

[6] The maximum relative error cited for the CSIN function is based upon a set of 2000 random arguments within the range. In the immediate proximity of the points $n\pi + 0i$ (where $n = \pm 1, \pm 2, \ldots,$) the relative error can be quite high, although the absolute error is small.

[7] The sample arguments were tangents of numbers uniformly distributed between $-\frac{\pi}{2}$ and $+\frac{\pi}{2}$.

[8] $x + iy = \delta e^{i\theta}$, where δ is exponentially distributed in $(0, 10^{35})$, and θ is uniformly distributed in $(-\pi, \pi)$.

[9] $x + iy = \delta e^{i\theta}$, where δ is exponentially distributed in $(0, 10^{75})$, and θ is uniformly distributed in $(-\pi, \pi)$.

[10] Tangents of linearly scaled random angles between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$.

[11] Accuracy figures are not available for the following entry names: FRXPR#, FRXPI#, FDXPD#, and FDXPI#.

Figure 48 (Part 6 of 6). Accuracy Figures

Appendix G. Assembler Language Information

The mathematical and service subprograms in the VS FORTRAN 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.
- Set up proper calling sequences.
- Supply correct parameters.

Library Availability

The assembler language programmer must arrange for the desired subprograms (modules) to be taken from the VS FORTRAN library and brought into main storage, usually as a part of the programmer's load module. This can be done by employing the techniques described in the appropriate publications for your operating system.

For example, in MVS, the VS FORTRAN library could be made part of the automatic call library for the linkage editor by using these job control statements:

```
//jobname JOB desired operands
//stepname EXEC ASMFCLG,PARM.LKED='XREF,LIST,MAP'
//ASM.SYSIN DD *

  (assembler language program source deck)

/*
//LKED.SYSLIB DD DSNAME=data set name,DISP=SHR
/*
```

Library subprograms 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 VS FORTRAN library as the data set name in the SYSLIB DD statement.

Calling Sequences

Two general methods of calling are possible:

- Code an appropriate macro instruction, such as CALL.
- Code assembler language branch instructions.

In all cases, a save area must be provided that:

- Is aligned on a fullword boundary
- Is 18 words in length
- Has its address in general register 13 at the time of the CALL macro instruction or branch

All extended precision mathematical subprograms (both explicit and implicit) use all 16 registers, and require their callers to supply a full 18-word save area.

Figure 55 on page 443 shows calling sequences for a specific example: how to find the square root of a value. The library square root subprogram (entry name SQRT) is invoked, using assembler language statements.

Figure 49 on page 438 through Figure 53 on page 441 contain assembler information for VS FORTRAN subprograms.

Assembler Information	**Figure**
Explicitly called mathematical subprograms	Figure 49
Implicitly called mathematical subprograms	Figure 50
Implicitly called character subprograms	Figure 51
Service subprograms	Figure 52
Explicitly called bit functions	Figure 53

Notes:

1. For performance reasons, VS FORTRAN subprograms use certain floating-point registers (see Figure 50 on page 440), but do not save and restore original register contents. If you want floating-point information retained, you must save it before calling the subprogram and restore it on return.

2. From the VSE control program, register 1 is not used, but the execution parameters are passed as bit settings in the communications area.

Assembler Language Calling Sequence

When a branch instruction, rather than a call macro instruction, is used to invoke a subprogram, several additional conventions must be observed:

- An argument (parameter) address list must be assembled on a fullword boundary. It consists of one 4-byte address constant for each argument, with the last address constant containing a 1 in its high order bit.
- The address of the first item in this argument address list must be in general register 1.
- From the VSE control program, register 1 is not used but the execution parameters are passed as bit settings in the communications area.
- The address of the entry point of the called subprogram must be in general register 15.
- The address of the point of return to the calling program must be in general register 14.

The total requirements for an assembler language calling sequence are illustrated in Figure 54 on page 441.

Supplying Correct Parameters

Arguments must be of the proper type, length, and quantity, and, in certain cases, within a specified range, for the subprogram called.

For mathematical and character subprograms, this information can be found in Figure 21 on page 288 through Figure 27 on page 300.

- INTEGER*4 denotes a signed binary number 4 bytes long.
- REAL*4, REAL*8, amd REAL*16 are normalized floating-point numbers 4, 8 and 16 bytes long, respectively.
- COMPLEX*8, COMPLEX*16, and COMPLEX*32 are complex numbers 8, 16 and 32 bytes long, respectively, whose first half contains the real part, and whose second half contains the imaginary part. Each part is a normalized floating-point number.
- Four-byte argument types must be aligned on fullword boundaries; 8-byte, 16-byte, and 32-byte types must be aligned on doubleword boundaries.

Argument information for nonmathematical subprograms can be found under Chapter 9, “Service and Utility Subroutines” on page 313.

Error messages resulting from incorrect arguments are explained in Appendix I, “Library Procedures and Messages” on page 463.

Mathematical Subprogram Results

Each mathematical subprogram returns a single answer of a type listed in Figure 21 on page 288 through Figure 27 on page 300.

- Integer answers are returned in general register 0.
- Real answers are returned in floating-point register 0.
- Complex answers are returned in floating-point registers 0 and 2.

Result registers are listed by subprogram entry name in Figure 49 on page 438 and Figure 50 on page 440.

For extended-precision mathematical subprograms, results are always returned in the floating-point registers:

- 0 and 2 for REAL*16 results
- 0, 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 9, "Service and Utility Subroutines" on page 313.

Space Considerations

Many of the mathematical subprograms require other mathematical subprograms for their calculations. In addition, most of the subprograms use the input/output, error processing, and interruption library subroutines. (This interdependence is outlined in Appendix E, "Storage Estimates" on page 417.) Thus, although you may request just one VS FORTRAN subprogram, the requirements of that subprogram may make the resultant load module quite large. The SQRT routine, for example, takes only 344 bytes of storage itself, but requires other subroutines that increase the load module size by approximately 20000 bytes.

Initializing the Execution Environment

If the called subprogram uses VS FORTRAN input/output, error processing, or interruption routines, the calling program must initialize the execution environment, as shown in Figure 54 on page 441 and Figure 55 on page 443.

An initialization entry to VFEIN# is not required if the main program is written in FORTRAN and the assembler language routine is a subroutine, because the VS FORTRAN compiler generates the initialization instructions in the FORTRAN main program, but does not generate them in a FORTRAN subroutine.

The initialization instructions cause a branch into the VFEIN# subprogram, which initializes return coding and prepares routines to handle interruptions. If this initialization is omitted, an interruption or error may cause abnormal termination. (After initialization, VFEIN# returns to the instruction following the BAL.)

Note: Before Release 4.0 of VS FORTRAN, the call to initialize the execution environment was made to VSCOM#, not VFEIN#. The call to VSCOM# is still supported, but you might significantly reduce the size of the load module by calling VFEIN# instead.

Terminating the execution environment is also recommended if VS FORTRAN input/output routines are used. This ensures that any partially filled output buffers are written to their data sets. The assembler language statements needed are those that the compiler generates for a STOP statement.

Subprogram Entry Name	Registers Used[1] Result	Intermediate
ABS,DABS,IABS,QABS	0	2
ACOS,ASIN	0	2,4
AIMAG,DIMAG,QIMAG	0	2,4,6
AINT,DINT,QINT	0	2,4,6
ALGAMA,GAMMA,LGAMMA	0	2,4,6
ALOG,LOG	0	2,4,6
ALOG10,LOG10	0	2,4,6
AMOD,DMOD,QMOD	0	2,4,6
ANINT,DNINT,NINT,IDNINT	0	2,4,6
ATAN,ATAN2	0	2,4,6
CABS	0,2	6
CCOS,CSIN	0,2	4
CDABS	0,2	4,6
CDCOS,CDSIN	0,2	4
CDEXP	0,2	4,6
CDLOG	0,2	4,6
CDSQRT	0,2	4,6
CEXP	0,2	4,6
CLOG	0,2	4,6
CONJG,DCONJG,QCONJG	0	2,4,6
COS	0	2,4
COSH,SINH	0	2,4
COTAN,TAN	0	2,4
CQABS	0,2,4,6	
CQCOS,CQSIN	0,2,4,6	
CQEXP	0,2,4,6	
CQLOG	0,2,4,6	
CQSQRT	0,2,4,6	
CSQRT	0,2	4,6
DACOS,DASIN	0	2,4
DATAN,DATAN2	0	2,4,6
DCOS	0	2,4
DCOS[2]	0	2,4,6
DCOSH,DSINH	0	2,4,6
DCOTAN,DTAN	0	2,4,6
DCOTAN,DTAN[2]	0	2,4,6
DDIM,DIM,IDIM,QDIM	0	2,4,6
DERF,DERFC	0	2,4,6
DEXP	0	2
DEXP[2]	0	2
DGAMMA,DLGAMA	0	2,4,6
DLOG	0	2,4,6
DLOG10	0	2,4,6
DPROD	0	2
DSIGN,ISIGN,SIGN,QSIGN	0	2,4,6
DSIN	0	2,4

Figure 49 (Part 1 of 2). Explicitly Called Mathematical Subprogram Assembler Information

DSIN[2]	0	2,4,6
DSQRT	0	2,4
DTANH	0	2,4,6
ERF,ERFC	0	2,4,6
EXP	0	
EXP[2]	0	
MOD	0[3]	
QARCOS,QARSIN	0,2	4,6
QATAN,QATAN2	0,2	4,6
QCOS,QSIN	0,2	4,6
QCOSH,QSINH	0,2	4,6
QCOTAN,QTAN	0,2	4,6
QERF,QERFC	0,2	4,6
QEXP,QLOG,QLOG10	0,2	4,6
QSQRT	0,2	4,6
QTANH	0,2	4,6
SIN	0	2,4
SQRT	0	2
TANH	0	2,4,6

[1] Floating-point
[2] Alternative mathematical library subroutines
[3] General register

Figure 49 (Part 2 of 2). Explicitly Called Mathematical Subprogram Assembler Information

Subprogram Entry Name	Registers Used[1] Result	Intermediate
CDDVD#	0,2	4,6
CDMPY#	0,2	4,6
CDVD#	0,2	4,6
CMPY#	0,2	4,6
CQDVD#,CQMPY#	0,2,4,6	
CXMPR#	0[3]	
FCDCD#	0,2,4,6	
FCDXI#	0,2	
FCQCQ#	0,2,4,6	
FCQXI#	0,2,4,6	
FCXPC#	0,2,4,6	
FCXPI#	0,2	
FDXPD#	0	
FDXPD#[2]	0	2,4,6
FDXPI#	0	
FIXPI#	0[3]	
FQXPI#	0,2	4,6
FQXPQ#,FQXP2#	0,2	4,6
FRXPI#	0	
FRXPR#	0	
FRXPR#[2]	0	2,4,6

[1] Floating-point
[2] Alternative mathematical library subroutines
[3] General register

Figure 50. Implicitly Called Mathematical Subprogram Assembler Information

Subprogram Entry Name(s)
CCMPR#
CHAR, ICHAR, LEN
CMOVE#
CNCAT#
INDEX
LGE, LGT, LLE, LLT

Figure 51. Implicitly Called Character Subprogram Assembler Information

Notes follow.

Subprogram Entry Name(s)
CDUMP, CPDUMP
DSPAN#, DSPN2#, DSPN4#
DUMP, PDUMP
DVCHK
DYCMN#
EXIT
OVERFL
SDUMP
XUFLOW

Figure 52. Service Subprogram Assembler Information

Subprogram Entry Name(s)
IBCLR, IBSET, BTEST, ISHFT
IOR, IEOR, NOT, IAND

Figure 53. Explicitly Called Bit Function Assembler Information

Notes to Figure 51, Figure 52, and Figure 53 :

No floating-point registers are used in:

- Implicitly called character subprograms
- Service subprograms
- Explicitly called bit functions

INSTRUCTION	EXPLANATION
USING RTN,15 RTN SAVE (14,12) DROP 15 LR 12,15 USING RTN,12 LR 15,13 LA 13,AREA ST 15,4(13) ST 13,8(15) LR 13,15	General register 13 contains the address of the save area.
SR 1,1 L 15,=V(VFEIN#) BALR 14,15	These statements are used to initialize the VS FORTRAN execution-time environment and may not be executed more than once. This initialization is necessary only when the main program is not written in FORTRAN.
LA 1,ARGLIST	General register 1 contains the address of the argument address list.
L 15,ENTRY	General register 15 contains the address of the subprogram.
BALR 14,15	General register 14 contains the address of the point of return to the calling program.
ENTRY DC V(entry name)	In the second case, the entry name must be defined by an EXTRN instruction to obtain proper linkage.
AREA DC 18F'0'	This statement defines the save area needed by the subprogram.
FOR ONE ARGUMENT:	
ARGLIST DC A(arg + X'80000000')	A 1 is placed in the high-order bit of the only argument.
FOR MORE THAN ONE ARGUMENT:	
ARGLIST DC A(ARG1)	This contains the address of the first argument.
DC A(ARG2) . . .	This contains the address of the second argument and any additional arguments.
DC A(ARGN + X'80000000') ARG1 DC F'1' ARG2 DC F'1' . . . ARGN DC F'1'	A 1 is placed in the high-order bit of the last argument.

Figure 54. General Assembler Language Calling Sequence

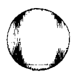
```
*         This shows the use of a CALL macro to
*         call the library square root subprogram.
          .
          .
          .
          CALL    SQRT,(AMNT),VL           (See Note 1)
          STE     0,ANSWER
          .
          .
          .
AMNT      DC      E'144'
ANSWER    DC      E'0'
          .
          .
          .
*         This shows the use of a BALR sequence to
*         call the library square root subprogram.
          .
          .
          .
          LA      1,ARG
          L       15,ENTRY
          BALR    14,15
          STE     0,ANSWER
          .
          .
          .
ENTRY     DC      V(SQRT)
ANSWER    DC      E'0'
          .
          .
          .
ARG       DC      A(AMNT+X'80000000')      (See Note 2)
          .
          .
          .
AMNT      DC      E'144'
```

Figure 55. Examples of Assembler Language Calling Sequences

Notes to Figure 55:

1. The VL operand in CALL indicates that the macro expansion should flag the end of the parameter list.

2. The CALL statement may not generate the same parameter list, nor one valid for MVS/XA.



Appendix H. Sample Storage Printouts

All output storage dumps are placed on the object error unit data set (defined by the installation during system generation).

A sample printout is shown in Figure 56 on page 446 for each dump format that can be specified for the storage dump subprogram using DUMP/PDUMP and CDUMP/CPDUMP. Figure 57 on page 447 shows the dump output using SDUMP.

```
CALL PDUMP WITH HEXADECIMAL FORMAT SPECIFIED

00A3E0     485F5E10  00000000  485F5E10  10000000  42100000

006DC8     42B00000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
006DF8     C0000000  00000000  41200000  41566666  0000000C  41100000

CALL PDUMP WITH LOGICAL*1 FORMAT SPECIFIED

006E1E     T  F

CALL PDUMP WITH LOGICAL*4 FORMAT SPECIFIED

006E10     F  T

CALL PDUMP WITH INTEGER*2 FORMAT SPECIFIED

006E18        10

006E1A      -100

006E1C        10

CALL PDUMP WITH INTEGER*4 FORMAT SPECIFIED

006E20              1          2          3          4          5          6          7          8          9         10
006E48             11         12

CALL PDUMP WITH REAL*4 FORMAT SPECIFIED

006E00      0.20000000E 01   0.53999996E 01

CALL PDUMP WITH REAL*8 FORMAT SPECIFIED

006DC8      0.17599999999999999D 03

CALL PDUMP WITH COMPLEX*8 FORMAT SPECIFIED

006DD0                  (3.0000000,4.0000000)              (4.0000000,8.0000000)

CALL PDUMP WITH COMPLEX*16 FORMAT SPECIFIED

006DE0               (0.9999999999999990,0.9999999999999990)           (-0.9999999999999990,-0.9999999999999990)

CALL PDUMP WITH LITERAL FORMAT SPECIFIED

006E5C     THIS ARRAY CONTAINS ALPHAMERIC DATA

CALL CPDUMP
008990     FILE READ ARGUMENT
```

Figure 56. Sample Storage Printout for DUMP/PDUMP and CDUMP/CPDUMP

Notes to Figure 56:

1. The headings on the printouts are not generated by the system, but were obtained by using FORMAT statements.

2. The number printed at the left of each output line is the storage location (in hexadecimal) of the first data item tabulated.

```
  CALL SDUMP WITH SCALAR VARIABLES OF VARIOUS TYPES

1SDUMP - SYMBOLIC DUMP FOR MODULE: MAIN
0 MODULE MAIN     WAS CALLED BY OP/SYS  .
0 MODULE MAIN     LAST CALLED IFYSDUMQ.
  FROM OFFSET 00033C AT ISN. NO.        30.
0         A  R8      0.55000000000000000D+01
-------------------------------------------------------------------------------
          P  L1      T
-------------------------------------------------------------------------------
         C1  C8      0.100000000E+01     0.100000000E+01
-------------------------------------------------------------------------------
         C3  C16     0.30000000000000000D+01     0.30000000000000000D+01
-------------------------------------------------------------------------------
        CH1  CHR
          1       F1                                  *1              *
-------------------------------------------------------------------------------
        CH2  CHR
          1       C1C2C3C4 C5C6C7C8                   *ABCDEFGH        *
-------------------------------------------------------------------------------
        JJJ  I2        222
-------------------------------------------------------------------------------
        LLL  I4          121212
-------------------------------------------------------------------------------
       IIII  I4            1111
-------------------------------------------------------------------------------
       YYYY  R16     0.25252525000000000000000000000000000Q+08
-------------------------------------------------------------------------------
       ZZZZ  R16     0.40000000000000000000000000000000000Q+06
-------------------------------------------------------------------------------
     IABLSE  I4               6
-------------------------------------------------------------------------------
     NUMLTS  I4               7
-------------------------------------------------------------------------------
 END OF SYMBOL DUMP PROCESSING
```

Figure 57. Sample Storage Printout for SDUMP

Output from Symbolic Dumps

When you call SDUMP, you receive output from the program units you specify. When your program abnormally terminates, you receive all variables that are in any program unit that has been processed, as well as those in the program unit currently being processed.

Output Format

In general, the output shows scalar items (one line only) and array (more than one line) items. Scalar and array items can contain either character or noncharacter data, but not both. The data is the same for both, except a top line is added for array items.

In addition, scalar and array items both identify valid variable types (shown as yyy in the formats).

Scalar Noncharacter

The scalar variable value printing scheme for noncharacter data is as follows:

```
xxxxxxxx yyy zzzzzzzz
```

where:

xxxxxxxx   is the variable name area.

yyy   is the variable type.

zzzzzzzz   is the area for the formatted output.

Valid variable types for yyy are:

I4 - Integer	(4 bytes)
I2 - Integer	(2 bytes)
L4 - Logical	(4 bytes)
L1 - Logical	(1 byte)
R4 - Real	(4 bytes)
R8 - Real	(8 bytes)
R16 - Real	(16 bytes)
C8 - Complex	(8 bytes, 4+4)
C16 - Complex	(16 bytes, 8+8)
C32 - Complex	(32 bytes, 16+16)
CHR - Character	

Scalar Character

The scalar variable value printing scheme for character data is as follows:

```
xxxxxxxx yyy
```

where:

xxxxxxxx   is the variable name.

yyy   is the type of the variable, followed by as many lines of the following form to display the entire variable value.

Character Data Format

The character data format is as follows:

```
xxxxxxxx   aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa *bbbbbbbbbbbbbbbb*
```

where:

xxxxxxxx	is the count of the next character displayed—the value is the decimal number of the character.
aaaaaaaa	is the hexadecimal representation of up to 4 bytes of character data—as many aa's are used as are needed to display the internal form of the data.
bbbbbbbbbbbbbbbb	is the EBCDIC representation of up to 16 bytes of character data—as many b's are used as are needed to display the data.

Note: Unprintable characters are translated to the character period (.), asterisks (*) are the delimiters of the EBCDIC character area.

Array

The array variable value printing scheme is as follows:

```
xxxxxxxx   yyy
```

where:

xxxxxxxx	is the array name.
yyy	is the array type.

Array Specification

```
DIMENSION x:  yyyyyyyy:zzzzzzzz aaa
```

where:

x	is the dimension (from 1 to 7).
yyyyyyyy	is the lower bound.
zzzzzzzz	is the upper bound.
aaa	is either blank or * ASSUMED SIZE ARRAY*.

Note: * ASSUMED SIZE ARRAY* appears only for the last dimension—there is one dimension line for each dimension of the array.

Array Data

For the display of data, the output is divided into two parts: Part 1 describes the array name and the current element indices, and part 2 displays the data.

The following shows how part 1 is formatted:

```
xxxxxxxx(dimensi1,dimensi2,dimensi3,dimensi4,dimensi5,
dimensi6,dimensi7)
```

where:

xxxxxxxx — is the array name.

dimensi1-dimensi7 — are the indexes of the element value to enclose the left and right parentheses, and the size of the variable areas are accurate. Only as many indexes as are needed are printed.

The data for part 2 is formatted in the same way as that for the scalar item already described, except that only the value is on the line.

A line of hyphens in the output marks the end of output for each scalar or array item (*not* an array element).

Array Message

The following message is issued if some array elements are missing:

```
MISSING ELEMENTS OF THE CURRENT ARRAY HAVE A VALUE OF ZERO
OR BLANK.
```

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:

xxxxxxxx — identifies the caller module.

yyyyyyyy — identifies the called routine.

OP/SYS — is the operating system: DOS, MVS, VS1, or CMS.

aaaaaa — is the offset into the program unit. If blanks appear, then the offset is not available.

bbbbbbbbbb — is the internal statement number (ISN). If double asterisks appear, the ISN information is unavailable.

Note: The message fragment is used in conjunction with other fragments to identify the CALLs and RETURNs of the program units.

```
MODULE xxxxxxxx LAST CALLED yyyyyyyy
```

where:

xxxxxxxx is the calling module name.

yyyyyyyy is the called module name.

Note: The message fragment is used in tracing the control flow of program units.

```
MODULE xxxxxxxx DID NOT CALL ANY OTHER ROUTINES.
```

where:

xxxxxxxx is the routine that did not call any other routines.

Note: The message fragment completes the group of fragments identifying the control flow scheme.

I/O Unit Information

The following messages appear only for post-ABEND processing (VPOST or VPOSA):

1. Default units

   DATA SET REFERENCE NUMBER TABLE. NUMBER OF ENTRIES IS xxx.
   Indicates the number of units available to the FORTRAN program is xxx.

   DEFAULT UNIT FOR THE PRINTER IS xxx.
   Indicates the default output device is xxx.

   DEFAULT UNIT FOR THE READER IS xxx.
   Indicates the default input device is xxx.

   DEFAULT UNIT FOR THE PUNCH IS xxx.
   Indicates the default punch output device is xxx.

   DEFAULT UNIT FOR THE OBJECT TIME ERROR MESSAGES IS xxx.
   Indicates that error messages issued by the VS FORTRAN program will go to unit xxx. This includes messages issued by IFYVPOST (abnormal termination) or IFYSDUMP (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 scalar items and the second for array items. Be aware, however, that if your program assigns values to both scalar 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. Scalar Items

This program assigns values to scalar items.

```
@PROCESS                                                                00000100
C                                                                       00000200
C     SAMPLE PROGRAM TO DEMONSTRATE SDUMP OF SCALAR VARIABLES           00000300
C                                                                       00000400
C                                                                       00000500
C     SPECIFY THE VARIABLE TYPES                                        00000600
C                                                                       00000700
      COMPLEX*8 C2,C1                                                   00000800
      COMPLEX*16 C3                                                     00000900
      CHARACTER CH1,CH2*8                                               00001100
      REAL*8 A,B                                                        00001200
      REAL*16 YYYY,ZZZZ                                                 00001300
      INTEGER*2 JJJ,KKK                                                 00001400
      LOGICAL*1 P                                                       00001500
      LOGICAL*4 Q                                                       00001600
C                                                                       00001700
C     ASSIGN THE VALUES                                                 00001800
C                                                                       00001900
      P=.TRUE.                                                          00002000
      Q=.FALSE.                                                         00002100
      CH2='ABCDEFGH'                                                    00002200
      CH1='1'                                                           00002300
      A = 5.5                                                           00002400
      NUMLTS=7                                                          00002500
      IABLSE=6                                                          00002600
      ZZZZ=4.0E5                                                        00002700
      IIII=1111                                                         00002800
      JJJ=222                                                           00002900
      OOO=327670                                                        00003000
                                                                        00003100

      PPP=32768                                                         00003200
                                                                        00003300
      B=222.222                                                         00003400
      KKK=999                                                           00003500
      LLL=121212                                                        00003600
      MMM=2147483647                                                    00003700
                                                                        00003800
      NNN=-2147483647                                                   00003900
                                                                        00004000
      YYYY=25252525                                                     00004100
      C1=(1.,1.)                                                        00004200
      C2=(2.,2.)                                                        00004300
      C3=(3.D0,3.D0)                                                    00004400
C                                                                       00004500
C     PRINT MESSAGE AND INVOKE SDUMP                                    00004600
C                                                                       00004700
      WRITE(6,*) ' CALL SDUMP WITH SCALAR VARIABLES OF VARIOUS TYPES '  00004800
      CALL SDUMP                                                        00004900
10    STOP                                                              00005000
      END                                                               00005100
```

Example 1. Output

The output is as follows:

```
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 IFYSDUMP.
 FROM OFFSET 00038C AT ISN. NO.       35.
        A  R8     0.55000000000000000D+01
--------------------------------------------------------------------------------
        B  R8     0.22222200012207031 2D+03
--------------------------------------------------------------------------------
        P  L1    T
--------------------------------------------------------------------------------
        Q  L4    F
-------------------------------------------------------------------------------
        R  R4     0.323733E+02
--------------------------------------------------------------------------------
       C1  C8     0.100000000E+01    0.100000000E+01
--------------------------------------------------------------------------------
       C2  C8     0.200000000E+01    0.200000000E+01
--------------------------------------------------------------------------------
       C3  C16    0.30000000000000000D+01    0.30000000000000000D+01
--------------------------------------------------------------------------------
      CH1  CHR
        1        F1                                     *1              *
--------------------------------------------------------------------------------
      CH2  CHR
        1        C1C2C3C4 C5C6C7C8                      *ABCDEFGH       *
--------------------------------------------------------------------------------
      JJJ  I2       222
--------------------------------------------------------------------------------
      KKK  I2       999
-------------------------------------------------------------------------------
      LLL  I4         121212
-------------------------------------------------------------------------------
      MMM  I4     2147483647
-------------------------------------------------------------------------------
      NNN  I4    -2147483647
-------------------------------------------------------------------------------
      OOO  I2     32767
-------------------------------------------------------------------------------
      PPP  I2     -32768
--------------------------------------------------------------------------------
     IIII  I4           1111
--------------------------------------------------------------------------------
     YYYY  R16    0.25252525000000000000000000000000Q+08
--------------------------------------------------------------------------------
     ZZZZ  R16    0.40000000000000000000000000000000Q+06
--------------------------------------------------------------------------------
   IABLSE  I4              6
--------------------------------------------------------------------------------
   NUMLTS  I4              7
--------------------------------------------------------------------------------
END OF SYMBOL DUMP PROCESSING FOR MAIN  .
```

Example 2. Array Items

This program assigns values to array items.

```
@PROCESS                                                                00000100
C                                                                       00000200
C     SAMPLE PROGRAM TO DEMONSTRATE SDUMP OF ARRAY VARIABLES            00000300
C                                                                       00000400
C                                                                       00000500
C     SPECIFY THE VARIABLE TYPES                                        00000600
C                                                                       00000700
      COMPLEX*8 C2(5),C1(5)                                             00000800
      COMPLEX*16 C3(5)                                                  00000900
      CHARACTER CH1(5),CH2(5)*8                                         00001100
      REAL*8 A(5),B(5)                                                  00001200
      REAL*16 YYYY(5),ZZZZ(5)                                           00001300
      INTEGER*2 JJJ(5),KKK(5)                                           00001400
      INTEGER NUMLTS(5),IABLSE(5),IIII(5),LLL(5)                        00001500
      LOGICAL*1 P(5)                                                    00001600
      LOGICAL*4 Q(5)                                                    00001700
C                                                                       00001800
C     ASSIGN THE VALUES                                                 00001900
C                                                                       00002000
      P(1)=.TRUE.                                                       00002100
      P(2)=.FALSE.                                                      00002200
      P(3)=.TRUE.                                                       00002300
      P(4)=.FALSE.                                                      00002400
      P(5)=.TRUE.                                                       00002500
      Q(1)=.TRUE.                                                       00002600
      Q(2)=.FALSE.                                                      00002700
      Q(3)=.FALSE.                                                      00002800
      Q(4)=.FALSE.                                                      00002900
      Q(5)=.TRUE.                                                       00003000
      CH2(1)(:)='ABCDEFGH'                                              00003100
      CH2(2)(:)='ABCIJKLM'                                              00003200
      CH2(3)(:)='ABCNOPQR'                                              00003300
      CH2(4)(:)='ABCSTUVW'                                              00003400
      CH2(5)(:)='ABCXYZAB'                                              00003500
      CH1(1)(:)='1'                                                     00003600
      CH1(2)(:)='2'                                                     00003700
      CH1(3)(:)='3'                                                     00003800
      CH1(4)(:)='4'                                                     00003900
      CH1(5)(:)='5'                                                     00004000
      A(1)  = 5.5                                                       00004100
      A(2)  = 4.5                                                       00004200
      A(3)  = 3.5                                                       00004300
      A(4)  = 2.5                                                       00004400
      A(5)  = 1.5                                                       00004500
      NUMLTS(1)=7                                                       00004600
      NUMLTS(2)=6                                                       00004700
      NUMLTS(3)=5                                                       00004800
      NUMLTS(4)=4                                                       00004900
      NUMLTS(5)=3                                                       00005000
      IABLSE(1)=6                                                       00005100
      IABLSE(2)=7                                                       00005200
      IABLSE(3)=8                                                       00005300
      IABLSE(4)=9                                                       00005400
      IABLSE(5)=10                                                      00005500
```

```
      ZZZZ(1)=4.0E5                                                     00005600
      ZZZZ(2)=4.0E3                                                     00005700
      ZZZZ(3)=4.0E2                                                     00005800
      ZZZZ(4)=4.0E1                                                     00005900
      ZZZZ(5)=4.0E0                                                     00006000
      IIII(1)=1111                                                      00006100
      IIII(2)=3211                                                      00006200
      IIII(3)=4311                                                      00006300
      IIII(4)=6511                                                      00006400
      IIII(5)=1541                                                      00006500
      JJJ(1)=212                                                        00006600
      JJJ(2)=242                                                        00006700
      JJJ(3)=232                                                        00006800
      JJJ(4)=252                                                        00006900
      JJJ(5)=262                                                        00007000
      B(1)=111.222                                                      00007100
      B(2)=222.222                                                      00007200
      B(3)=333.222                                                      00007300
      B(4)=444.222                                                      00007400
      B(5)=555.222                                                      00007500
      KKK(1)=899                                                        00007600
      KKK(2)=799                                                        00007700
      KKK(3)=699                                                        00007800
      KKK(4)=599                                                        00007900
      KKK(5)=499                                                        00008000
      LLL(1)=212                                                        00008100
      LLL(2)=312                                                        00008200
      LLL(3)=412                                                        00008300
      LLL(4)=512                                                        00008400
      LLL(5)=612                                                        00008500
      YYYY(1)=15151515                                                  00008600
      YYYY(2)=25252525                                                  00008700
      YYYY(3)=35353535                                                  00008800
      YYYY(4)=45454545                                                  00008900
      YYYY(5)=55555555                                                  00009000
      C1(1)=(5.,1.)                                                     00009100
      C1(2)=(4.,2.)                                                     00009200
      C1(3)=(3.,3.)                                                     00009300
      C1(4)=(2.,4.)                                                     00009400
      C1(5)=(1.,5.)                                                     00009500
      C2(1)=(2.,10.)                                                    00009600
      C2(2)=(4.,8.)                                                     00009700
      C2(3)=(6.,6.)                                                     00009800
      C2(4)=(8.,4.)                                                     00009900
      C2(5)=(10.,2.)                                                    00010000
      C3(1)=(3.D0,13.D0)                                                00010100
      C3(2)=(6.D0,11.D0)                                                00010200
      C3(3)=(9.D0,9.D0)                                                 00010300
      C3(4)=(12.D0,7.D0)                                                00010400
      C3(5)=(15.D0,5.D0)                                                00010500
C                                                                       00011100
C     PRINT MESSAGE AND INVOKE SDUMP                                    00011200
C                                                                       00011300
      WRITE(6,*) ' CALL SDUMP WITH ARRAY VARIABLES OF VARIOUS TYPES '   00011400
      CALL SDUMP                                                        00011500
      STOP                                                              00011600
      END                                                               00011700
```

Example 2. Output

The output is as follows:

```
CALL SDUMP WITH ARRAY VARIABLES OF VARIOUS TYPES

SDUMP - SYMBOLIC DUMP FOR MODULE: MAIN
 MODULE MAIN     WAS CALLED BY OP/SYS  .
 MODULE MAIN     LAST CALLED IFYSDUMP.
 FROM OFFSET 000C08 AT ISN. NO.        103.
 ARRAY:        A   TYPE:R8
   DIMENSION 1:        1:       5
        A(       1)   0.55000000000000000D+01
        A(       2)   0.45000000000000000D+01
        A(       3)   0.35000000000000000D+01
        A(       4)   0.25000000000000000D+01
        A(       5)   0.15000000000000000D+01
--------------------------------------------------------------------------------
 ARRAY:        B   TYPE:R8
   DIMENSION 1:        1:       5
        B(       1)   0.111222000122070312D+03
        B(       2)   0.222222000122070312D+03
        B(       3)   0.333221923828125000D+03
        B(       4)   0.444221923828125000D+03
        B(       5)   0.555221923828125000D+03
--------------------------------------------------------------------------------
 ARRAY:        P   TYPE:L1
   DIMENSION 1:        1:       5
        P(       1)  T
        P(       2)  F
        P(       3)  T
        P(       4)  F
        P(       5)  T
--------------------------------------------------------------------------------
 ARRAY:        Q   TYPE:L4
   DIMENSION 1:        1:       5
        Q(       1)  T
        Q(       2)  F
        Q(       3)  F
        Q(       4)  F
        Q(       5)  T
--------------------------------------------------------------------------------
 ARRAY:       C1   TYPE:C8
   DIMENSION 1:        1:       5
       C1(       1)   0.500000000E+01  0.100000000E+01
       C1(       2)   0.400000000E+01  0.200000000E+01
       C1(       3)   0.300000000E+01  0.300000000E+01
       C1(       4)   0.200000000E+01  0.400000000E+01
       C1(       5)   0.100000000E+01  0.500000000E+01
--------------------------------------------------------------------------------
 ARRAY:       C2   TYPE:C8
   DIMENSION 1:        1:       5
       C2(       1)   0.200000000E+01  0.100000000E+02
       C2(       2)   0.400000000E+01  0.800000000E+01
       C2(       3)   0.600000000E+01  0.600000000E+01
       C2(       4)   0.800000000E+01  0.400000000E+01
       C2(       5)   0.100000000E+02  0.200000000E+01
--------------------------------------------------------------------------------
```

```
ARRAY:       C3    TYPE:C16
  DIMENSION 1:          1:        5
      C3(        1)  0.300000000000000000D+01  0.130000000000000000D+02
      C3(        2)  0.600000000000000000D+01  0.110000000000000000D+02
      C3(        3)  0.900000000000000000D+01  0.900000000000000000D+01
      C3(        4)  0.120000000000000000D+02  0.700000000000000000D+01
      C3(        5)  0.150000000000000000D+02  0.500000000000000000D+01
------------------------------------------------------------------------------
ARRAY:      CH1    TYPE:CHR
  DIMENSION 1:          1:        5
     CH1(        1)
       1       F1                                      *1          *
     CH1(        2)
       1       F2                                      *2          *
     CH1(        3)
       1       F3                                      *3          *
     CH1(        4)
       1       F4                                      *4          *
     CH1(        5)
       1       F5                                      *5          *
------------------------------------------------------------------------------
ARRAY:      CH2    TYPE:CHR
  DIMENSION 1:          1:        5
     CH2(        1)
       1       C1C2C3C4 C5C6C7C8                       *ABCDEFGH        *
     CH2(        2)
       1       C1C2C3C9 D1D2D3D4                       *ABCIJKLM        *
     CH2(        3)
       1       C1C2C3D5 D6D7D8D9                       *ABCNOPQR        *
     CH2(        4)
       1       C1C2C3E2 E3E4E5E6                       *ABCSTUVW        *
     CH2(        5)
       1       C1C2C3E7 E8E9C1C2                       *ABCXYZAB        *
------------------------------------------------------------------------------
ARRAY:      JJJ    TYPE:I2
  DIMENSION 1:          1:        5
     JJJ(        1)      212
     JJJ(        2)      242
     JJJ(        3)      232
     JJJ(        4)      252
     JJJ(        5)      262
------------------------------------------------------------------------------
ARRAY:      KKK    TYPE:I2
  DIMENSION 1:          1:        5
     KKK(        1)      899
     KKK(        2)      799
     KKK(        3)      699
     KKK(        4)      599
     KKK(        5)      499
------------------------------------------------------------------------------
```

```
ARRAY:      LLL    TYPE:I4
  DIMENSION 1:          1:        5
     LLL(          1)           212
     LLL(          2)           312
     LLL(          3)           412
     LLL(          4)           512
     LLL(          5)           612
--------------------------------------------------------------------------------
ARRAY:     IIII    TYPE:I4
  DIMENSION 1:          1:        5
    IIII(          1)          1111
    IIII(          2)          3211
    IIII(          3)          4311
    IIII(          4)          6511
    IIII(          5)          1541
--------------------------------------------------------------------------------
ARRAY:     YYYY    TYPE:R16
  DIMENSION 1:          1:        5
    YYYY(          1)    0.151515150000000000000000000000000Q+08
    YYYY(          2)    0.252525250000000000000000000000000Q+08
    YYYY(          3)    0.353535350000000000000000000000000Q+08
    YYYY(          4)    0.454545450000000000000000000000000Q+08
    YYYY(          5)    0.555555550000000000000000000000000Q+08
--------------------------------------------------------------------------------
ARRAY:     ZZZZ    TYPE:R16
  DIMENSION 1:          1:        5
    ZZZZ(          1)    0.400000000000000000000000000000000Q+06
    ZZZZ(          2)    0.400000000000000000000000000000000Q+04
    ZZZZ(          3)    0.400000000000000000000000000000000Q+03
    ZZZZ(          4)    0.400000000000000000000000000000000Q+02
    ZZZZ(          5)    0.400000000000000000000000000000000Q+01
--------------------------------------------------------------------------------
ARRAY:   IABLSE    TYPE:I4
  DIMENSION 1:          1:        5
  IABLSE(          1)             6
  IABLSE(          2)             7
  IABLSE(          3)             8
  IABLSE(          4)             9
  IABLSE(          5)            10
--------------------------------------------------------------------------------
ARRAY:   NUMLTS    TYPE:I4
  DIMENSION 1:          1:        5
  NUMLTS(          1)             7
  NUMLTS(          2)             6
  NUMLTS(          3)             5
  NUMLTS(          4)             4
  NUMLTS(          5)             3
--------------------------------------------------------------------------------
END OF SYMBOL DUMP PROCESSING FOR MAIN    .
```

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 doesn't belong to the program.

```
DIMENSION A(10)                                             00000900
A(5)=3.2                                                    00001000
I=99999999                                                  00001400
A(I)=2.3                                                    00001800
STOP                                                        00001900
END                                                         00002000
```

The output from running this program is shown on the next page. The output you get will vary, depending on the compiler options GOSTMT/NOGOSTMT, SDUMP/NOSDUMP, and TEST/NOTEST, as follows:

- All lines of the IFY240I message except the last line are option independent. The last IFY240I line contains ISNs/line numbers when the abending module was compiled with SDUMP or TEST; otherwise, it contains ** in those fields. For details on IFY240I, see Appendix I, "Library Procedures and Messages" on page 463.

- 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 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 451.

- Control flow information appears last and is dependent on the SDUMP or TEST option. For a program unit active at abend that was compiled with SDUMP or TEST, the control flow information contains ISNs/line numbers; otherwise, it contains ** in those fields. For details on control flow, see "Control Flow Information" on page 450.

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.

Example 3. Output

The output from the failing program is as follows:

```
IFY240I VSTAE - ABEND CODE IS: SYSTEM 00C5, USER 0000. SCB/SDWA=00033178.
IFY240I VSTAE - IO HALTED.     PSW=FFE4000582020110. ENTRY POINT=00020000.
IFY240I VSTAE - REGS  0-3  6BA141C0 000000C5 0002012E 00020050
IFY240I VSTAE - REGS  4-7  000125E0 00000006 17D783FC 0C000878
IFY240I VSTAE - REGS  8-11 00018E38 00019E38 40404040 00002A30
IFY240I VSTAE - REGS 12-15 600190FC 00020050 80020136 000200F0
IFY240I VSTAE - FRGS  0-3  4124CCCD 00000000 00000000 00000000
IFY240I VSTAE - FRGS  4-7  00000000 00000000 00000000 00000000
IFY240I VSTAE - ABEND IN MODULE MAIN AT ISN 4 (OFFSET 000001FC).

TRACEBACK OF CALLING ROUTINES; MODULE ENTRY ADDRESS=020000
 MAIN    (020000) CALLED BY OPERATING SYSTEM.

DATA SET REFERENCE NUMBER TABLE. NUMBER OF ENTRIES IS 100.
DEFAULT UNIT FOR THE OBJECT TIME ERROR MESSAGES IS   6.
DEFAULT UNIT FOR THE READER IS   5.
DEFAULT UNIT FOR THE PRINTER IS   6.
DEFAULT UNIT FOR THE PUNCH IS   7.
FILE ON UNIT   6 IS ACTIVE.
 FILE IS USED FOR SYNCHRONOUS SEQUENTIAL I/O.
 FILE STATUS IS OLD.
 FILE IS FORMATTED.
 FILE NAME USED IS FT06F001.
VPOSA - POST ABEND SYMBOLIC DUMP FOR MODULE: MAIN
 MODULE MAIN     WAS CALLED BY OP/SYS   .
 MODULE MAIN     LAST CALLED VFEIM# .
 FROM OFFSET  000212.

 ARRAY:       A   TYPE:R4
   DIMENSION 1:        1:       10
        A(       5)   0.320000E+01
MISSING ELEMENTS OF THE CURRENT ARRAY HAVE A VALUE OF ZERO OR BLANK
---------------------------------------------------------------------
        I  I4       99999999
---------------------------------------------------------------------
END OF SYMBOL DUMP PROCESSING FOR MAIN   .

PROGRAM UNIT NOT COMPILED FOR SYMBOLIC DUMP PROCESSING. PROGRAM UNIT IS
 VFEIM#
 MODULE VFEIM#   WAS CALLED BY MAIN     .
 FROM OFFSET 000212.
 MODULE VFEIM#   LAST CALLED IFYVPARM.
 FROM OFFSET 0003D4.

PROGRAM UNIT NOT COMPILED FOR SYMBOLIC DUMP PROCESSING. PROGRAM UNIT IS
 IFYVPARM
 MODULE IFYVPARM   WAS CALLED BY VFEIM#.
 FROM OFFSET 0003D4.
 MODULE IFYVPARM   LAST CALLED IFYVSPIE.
 FROM OFFSET 00017A.

PROGRAM UNIT NOT COMPILED FOR SYMBOLIC DUMP PROCESSING. PROGRAM UNIT IS
 IFYVSPIE.
```

Appendix I. Library Procedures and Messages

This appendix contains explanations of the program-interruption and error procedures used by the VS FORTRAN 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*. For detailed information about error processing and message formats, see *VS FORTRAN Programming Guide*.

Library Interruption Procedures

The VS FORTRAN Library processes those interrupts that are described below; all others are handled directly by the system supervisor:

1. When an interrupt occurs, indicators are set to record exponent overflow, underflow, fixed-point, floating-point, or decimal divide exceptions. These indicators can be interrogated dynamically by the subprograms described under Chapter 9, "Service and Utility Subroutines" on page 313.

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) occur. 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 21 through Figure 26, 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 Library. This facility provides for standard corrective action by the user. (For a full description of extended error handling, see *VS FORTRAN Programming Guide.*)

Library Messages

The VS FORTRAN Library generates three types of messages:

- Program-interrupt messages
- Execution error messages
- Operator messages

All library messages are numbered. Program-interrupt messages are written when an exception to a system restriction occurs, such as dividing by zero or generating a result too large to contain in a floating point register. Execution error messages are written when a VS FORTRAN Library function or subroutine is misused or an I/O error occurs. Operator messages are written when a STOP n or PAUSE statement is executed.

Except for operator and informational messages, all VS FORTRAN 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 FORTRAN 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:

  ```
  LAST EXECUTED FORTRAN STATEMENT IN PROGRAM name AT
  SEQ. NO. ssssss (OFFSET oooooooo).
  ```

- Program unit compiled with SDUMP or, for some errors, GOSTMT:

  ```
  LAST EXECUTED FORTRAN STATEMENT IN PROGRAM name AT
  ISN iiiiii (OFFSET oooooooo).
  ```

where:

name is the name of failing program unit (reentrant CSECT name if compiled with RENT).

oooooooo is the hexadecimal offset from the beginning of the program to the last-executed statement.

ssssss is the sequence number of source statement in failing program unit.

iiiiii is the compiler-generated internal sequence 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 which invoked the VS FORTRAN 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 VS FORTRAN program units are encountered in the active chain of program units that caused issuance of the error message.

See *VS FORTRAN Compiler and Library Diagnosis Guide* if a problem recurs after you have performed the specified programmer response for the message received.

Program-Interrupt Messages

Program-interrupt messages are written with the old Program Status Word (PSW), which aids the programmer in determining the nature of the error.

There are four program-interrupt messages: IFY207I, IFY208I, IFY209I, and IFY210I. The messages are issued by a library module. The last five characters in the name—**VFNTH** for OS/VS and CMS systems, **DFNTH** for VSE systems—appears after the identifier of each message. Here are the messages and their explanations:

IFY207I VFNTH : PROGRAM INTERRUPT (Z) - FLOATING-POINT EXCEPTION OVERFLOW, REGISTER CONTAINS nnnnnnnn

Explanation: This message indicates that an overflow exception, identified by the character 8 or C in the 8th position of the PSW, has occurred. This exception occurs for a fixed-point overflow either when a carry occurs out of the high-order bit position in fixed-point arithmetic operations, or when high-order significant bits are lost during the algebraic left-shift operation. For an exponent overflow, this exception occurs when the magnitude of the result operation is greater than or equal to 16^{63} (approximately 7.2×10^{75}).

Supplemental Data Provided: The floating-point number (nnnnnnnn) before alteration for an exponent-overflow exception.

Standard Corrective Action: Execution continues at the point of the interrupt. For an exponent overflow, 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 ($16^{63}*(1-16^{-28})$)

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.

IFY208I VFNTH : PROGRAM INTERRUPT (Z) - FLOATING-POINT UNDERFLOW EXCEPTION, PSW xxxxxxxxxxxxxxxx REGISTER CONTAINS nnnnnnnn

Explanation: The message indicates that an exponent-underflow exception, identified by a D in the 8th position of the PSW, has occurred. This exception occurs when the result of a floating-point arithmetic operation is less than 16^{-65} (approximately 5.4×10^{-79}).

Supplemental Data Provided: The floating-point number (nnnnnnnn) before alteration.

Standard Corrective Action: Execution continues at the point of the interrupt, with the result register 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.

IFY209I VFNTH : PROGRAM INTERRUPT (Z) - yyyyyy EXCEPTION, PSW xxxxxxxxxxxxxxxx REGISTER CONTAINS nnnnnnnn

Explanation: This message indicates that an attempt to divide by 0 has occurred. A fixed-point-divide exception is identified by a 9 in the 8th position of the PSW; a floating-point-divide exception, by an F.

Supplemental Data Provided: Floating-point number (nnnnnnnn) before alteration, for a floating-point interrupt.

Standard Corrective Action: For floating-point-divide, execution continues at the point of the interrupt with the result registers set to:

- True zero of correct precision for case of n/0, where n=0.
- Largest possible floating-point number of correct precision for case of n/0 where n≠0. For fixed-point-divide, leave registers unmodified and continue execution.

Programmer Response: Either correct the source where division by 0 is occurring, or modify previous source statements to test for the possibilities, or bypass the invalid division.

IFY210I VFNTH : PROGRAM INTERRUPT (Z) - yyyyyy EXCEPTION, PSW xxxxxxxxxxxxxxxx

Explanation: A program interruption occurred. As indicated by the "P" the interruption was precise; that is, it was not a specification exception. (A specification exception may nevertheless occur on machines that allow imprecise interruptions.)

Standard Corrective Action: The operation is suppressed and message IFY240I is issued on MVS and VM systems. On DOS systems, the operation is suppressed and the partition is dumped, if abend dumps were requested.

Supplemental Data Provided: The type of interruption (cccccccccccc) and the PSW at the time of the interruption (xxxxxxxxxxxxxxxx).

The type of interruption will be one of the following:

Operation exception
Privileged-operation exception
Execute exception
Protection exception
Addressing exception
Specification exception
Data exception
Fixed-point-overflow exception
Fixed-point-divide exception
Decimal-overflow exception
Decimal-divide exception
Exponent-overflow exception
Exponent-underflow exception
Significance exception
Floating-point-divide exception

The causes of these interruptions are explained in *IBM System/370 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 overwrote 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 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 VS FORTRAN 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.

Execution Error Messages

Each of these has the form:

IFYxxxI zzzzz : message text

where:

xxx is the number of the library message.

zzzzz is the last five characters of the module named IFYzzzzz.

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.

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.

IFY120I DKIOS | VKIOS : OPEN STATEMENT ATTEMPTED TO CHANGE pppppppp FOR FILE ffffff WHICH IS ALREADY OPEN. ONLY 'BLANK' MAY BE CHANGED.

Explanation: An OPEN statement was issued for a file that is already open. The OPEN statement contains a parameter whose value has already been set and cannot be changed. When a file is already open, only the BLANK parameter can be specified on an OPEN statement.

Supplemental Data Provided:

pppppppp is ACCESS, FORM, ACTION, KEYS, or PASSWORD—the parameter on the OPEN statement whose value you cannot change.

fffffff is the name of the file that is already open.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: If you want to change the value of the BLANK parameter, first remove the parameter that should not have been specified. Otherwise, remove the OPEN statement or open a different file with it.

IFY121I **DKIOS | VKIOS : OPEN STATEMENT FOR FILE ffffff SPECIFIES ACTION='WRITE' BUT HAS MORE THAN ONE KEY IN 'KEYS' PARAMETER.**

Explanation: An OPEN statement has conflicting parameters: ACTION='WRITE', which implies you are loading a file, and KEYS with more than one key listed.

Supplemental Data Provided: ffffff is the name of the file you tried to open.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: If you want to load the file, remove the KEYS parameter 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 parameter to READ or READWRITE.

IFY122I **DKIOS | VKIOS : ssssssss STATEMENT IS NOT ALLOWED WHEN THE FILE IS OPEN WITH AN ACTION OF 'dddddddd'. FILE fffffff.**

Explanation: The value of the ACTION parameter on an OPEN statement conflicts with a statement that follows the OPEN statement. For an ACTION parameter with the value of WRITE, only the WRITE and CLOSE statements are allowed. For a value of READ, no update statement (WRITE, REWRITE, DELETE) is allowed.

Supplemental Data Provided:

ssssssss is the name of the incompatible statement.

dddddddd is the value of the ACTION parameter that is in use.

fffffff is the name of the file.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the ACTION parameter or remove the incompatible statement.

IFY123I **DKIOS | VKIOS : ssssssss STATEMENT IS NOT ALLOWED FOLLOWING ssssssss STATEMENT WHICH RESULTED IN cccccccccccccccc CONDITION. FILE fffffff.**

Explanation: A statement was not allowed because a previous statement caused an error and the loss of position in the file being processed. You cannot read records sequentially or use a BACKSPACE, DELETE, or REWRITE statement until you have re-established file position.

Supplemental Data Provided:

The first ssssssss is the name of the statement that was not allowed.

The second ssssssss is the name of the earlier statement that caused the error.

cccccccccccccccc is RECORD NOT FOUND, DUPLICATE ERROR, END OF FILE, VSAM I/O ERROR, or PROGRAM LOGIC ERROR.

fffffff is the name of the file.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Code either a REWIND or 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.

IFY124I DKIOS | VKIOS : READ STATEMENT SPECIFIES 'KEYID' VALUE OF nnnnn WHICH CONFLICTS WITH 'KEYS' PARAMETER ON OPEN STATEMENT FOR FILE fffffff.

Explanation: The value of the KEYID parameter is larger than the number of start-end pairs in the KEYS parameter. Therefore no pair (and hence no key) can be associated with the KEYID parameter. This conflict can arise even if no KEYS parameter is coded: a default of one key is assumed, so if KEYID has a value greater than 1, an error exists.

Supplemental Data Provided:

nnnnn is the value of the KEYID parameter on the READ statement.

fffffff is the name of the file for which the READ statement was issued.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the KEYID parameter so that it is no larger than the number of start-end pairs in the KEYS parameter, or remove the KEYID parameter.

IFY125I DKIOS | 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 parameter of a READ statement. This argument is longer than the key being searched for.

Supplemental Data Provided:

nnnnnnnn is the length in bytes of the search (or key) argument.

k is the relative position in a list of keys of the key of reference—the key currently in use. The list of keys is in the KEYS parameter of the OPEN statement. ("KEYID IS k" is omitted if the KEYS parameter of the OPEN statement specifies only one key or was not coded.)

mmmmmmmm is the length in bytes of the key being used.

fffffff is the name of the file for which the READ statement was issued.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Specify a search argument in the KEY, KEYGE, or KEYGT parameter 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 parameter.

IFY126I DKIOS | VKIOS : RECORD NOT FOUND WITH SPECIFIED KEY. FILE fffffff. [KEYID IS k: sssss:eeeee.] xxxxx PARAMETER IS vvvvvvvvvv

Explanation: There was no record in the file meeting the search argument in the KEY, KEYGE, or KEYGT parameter of the READ statement. The search was based on the key specified in the KEYID parameter of the READ statement. (If there was no KEYID parameter in the READ statement, the search was based on the KEYID parameter last used. If no KEYID parameter has been used since the file was opened, the first key specified in the KEYS parameter of the OPEN statement was used for the search.)

Supplemental Data Provided:

fffffff is the name of the file for which the READ statement was issued.

k is the relative positon in a list of keys of the key of reference—the key currently in use. The list of keys is in the KEYS parameter of the OPEN statement. (This part of the message and the sssss:eeeee information are omitted if the KEYS parameter of the OPEN statement specified only one key or was not coded.)

sssss is the starting position in each record of the key being used, and eeeee is the ending position.

xxxxx is KEY, KEYGE, or KEYGT—whichever parameter was used in the READ statement.

vvvvvvvvvv is the value of the parameter.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the KEY, KEYGE, or KEYGT parameter 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 parameter to your program. It specifies the statement to be given control when this condition occurs.

IFY127I VIOUF | VIOFM : THE VSAM KSDS ssssssss STATEMENT REFERS TO UNIT nn WHICH IS NOT OPEN

Explanation: An input/output statement referred to a unit that was not opened with an OPEN statement.

Supplemental Data Provided:

ssssssss is the name of the input/output statement—for example, READ, REWRITE, DELETE.

nn is the unit number referred to in the input/output statement.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Change the program to issue an OPEN statement with the ACCESS='KEYED' parameter before issuing the input/output statement.

IFY128I DKIOS | VKIOS : THE VSAM KSDS ssssssss STATEMENT REFERS TO FILE fffffff WHICH IS NOT A VSAM KSDS FILE.

Explanation: An input/output statement was issued that can apply only to a VSAM file. The file, however, was opened as a non-VSAM file.

Supplemental Data Provided:

ssssssss is the name of the input/output statement.

fffffff is the name of the file.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter 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.

IFY129I DKIOS | VKIOS : THE VSAM KSDS RECORD SUPPLIED BY THE ssssssss STATEMENT HAD A LENGTH OF 0. FILE fffffff.
IFY129I DKIOS | VKIOS : THE VSAM KSDS 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 parameter of the OPEN statement or implied by the operating system's data definition statement).

Supplemental Data Provided:

sssssssss is either WRITE or REWRITE.

nnnnn is the length of the record that was built.

fffffff is the name of the file involved in the input/output operation.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter 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.

IFY130I **DKIOS | VKIOS : ERROR ON VSAM KSDS WHILE PROCESSING sssssssss 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:

sssssssss is the name of the statement being processed.

fffffff is the name of the file involved in the input/output operation.

mmmmmm is the name of the VSAM macro that was issued (GET, PUT, POINT, and so on).

rc is the VSAM return code.

hc is the VSAM error feedback code in hexadecimal.

dc is the same code in decimal.

fc is the function code in hexadecimal.

You can find an explanation of the codes in either *VSE/VSAM messages and Codes* or *OS/VS Virtual Storage Access Method (VSAM) Programmer's Guide*.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Take the action given in the appropriate manual.

IFY131I **DKIOS | VKIOS : DUPLICATE FILE NAME WOULD BE GENERATED FOR FILE fffffff SINCE THERE ARE k KEYS SPECIFIED IN 'KEYS' PARAMETER OF THE OPEN STATEMENT.**

Explanation: When opening files for multiple-key processing, VS FORTRAN 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 maximum length of 7

characters, a number cannot be appended, so the last character is overlayed 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 proceeded to generate file names, it would duplicate the file name given in the OPEN statement.

Supplemental Data Provided:

ffffff is the name of the file.

k is the number of key specified in the KEYS parameter of the OPEN statement.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter 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.

IFY132I DKIOS | VKIOS : FILE fffff 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 found that the files had different maximum record lengths. Therefore, the data definition statements for the files must contain an error or inconsistency. For example, a statement may refer to an alternate-index file rather than to a *path* from the alternate-index file to the base cluster. Or statements may point to alternate-index files for different base clusters. Or they may mistakenly refer to two base clusters and no alternate-index files.

Supplemental Data Provided:

ffffff is the file name.

f2 is the related file.

R is the record length of the file.

R2 is the record length of the related file.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Change the data definition statements (DD statements in OS/VS, DLBL statements in VM and VSE) to refer to the VSAM files that represent the same base cluster.

IFY133I 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 found that one of the files was empty.

Supplemental Data Provided: fffffff is the ddname of the empty file.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter 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.

IFY134I DKIOS | VKIOS : OPEN STATEMENT FOR VSAM KSDS FILE fffffff SPECIFIES A KEY OF sssss:eeeee, BUT THERE IS NO FILE AVAILABLE WITH THIS KEY.

Explanation: A key specified on the OPEN statement does not correspond to any of the files, specified by ddnames, that were opened for keyed access.

Supplemental Data Provided:

fffffff is the name of the file, specified explicitly or taken by default, in the OPEN statement.

sssss is the starting position in each record of the key to be used; eeeee is the ending position.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Correct the starting and ending positions of the keys in the KEYS parameter; each key must correspond to a file that is identified in a data definition statement (a DD statement in OS/VS, a DLBL statement in VM and VSE). (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 parameter of the access method services DEFINE command. There the first position in a record is position 0.

IFY135I DKIOS | VKIOS : ATTEMPT MADE TO ADD A RECORD WITH A DUPLICATE KEY TO A VSAM KSDS. FILE fffffff. THE KEY OF REFERENCE HAS A KEYID OF k, A POSITION OF sssss:eeeee, AND A VALUE OF vvvvvvvvv (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 parameter of the OPEN statement for the file.

Supplemental Data Provided:

ffffff is the name of the file.

k indicates the key of reference—that is, the start-end pair in the KEYS parameter of the file's OPEN statement that was used in writing the record.

sssss:eeeee is the position in the record of the key of reference.

vvvvvvvvvv is the value of the key of reference.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter 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 parameter. It identifies the statement to be given control if the condition occurs.

IFY136I VOPEN : ACTION PARAMETER IS NOT VALID FOR AN OPEN STATEMENT. UNIT nn.

Explanation: The ACTION parameter on the OPEN statement specified a value other than READ, WRITE, or READWRITE.

Supplemental Data Provided: nn is the unit number specified in the OPEN statement.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter was coded, control is passed to the indicated statement.

Programmer Response: Change the value of the ACTION parameter to READ, WRITE, or READWRITE.

IFY137I VOPEN : KEYS PARAMETER ON AN OPEN STATEMENT IS NOT ALLOWED EXCEPT WITH ACCESS='KEYED'. UNIT nn.

Explanation: The OPEN statement has a KEYS parameter, but has either no ACCESS parameter or one whose value is incompatible with KEYS. (Only the value KEYED is compatible.)

Supplemental Data Provided: nn is the unit number specified on the OPEN statement.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter 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 parameter from the OPEN statement.

IFY138I DKIOS | VKIOS : ATTEMPTED TO OPEN AN EMPTY VSAM KSDS WITH ACTION='READ'. VSAM ERROR CODE X'6E' (110). FILE fffffff.

Explanation: VSAM does not allow an empty file to be opened for input operations.

Supplemental Data Provided: ffffff is the name of the file for which the OPEN statement was issued.

Standard Corrective Action: Execution continues, but the I/O request is ignored. If the ERR parameter 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.

IFY139I DKIOS | VKIOS : ATTEMPT MADE TO REWRITE A RECORD IN WHICH THE VALUE OF THE KEY OF REFERENCE DIFFERS FROM THE VALUE OF THAT KEY IN THE RECORD JUST READ. THE KEY OF REFERENCE HAS A KEYID OF k. FILE fffffff.

Explanation: You read a record and, in trying to rewrite it, wrote a key of reference whose value differed from that in the original record.

Supplemental Data Provided: The name of the file (fffffff) and, if the file has multiple keys, the KEYID (k) of the key of reference.

Standard Corrective Action: Execution continues, but the I/O request is ignored.

Programmer Response: If you did not intend to write a new key value, make sure that:

- The I/O list contains all the fields of the record to be rewritten
- Changes in the order or length of nonkey 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.

IFY140I DKIOS | VKIOS : KEY SEQUENCE ERROR LOADING A VSAM KSDS. FILE fffffff. THE KEY OF REFERENCE IN THE REJECTED RECORD HAD A VALUE OF vvvvvvvvv.

Explanation: You attempted to load a record in which the value of the primary key was not greater than the value of the primary key in the previous record.

Supplemental Data Provided: The name of the file (fffffff) and the value of the key of reference in the record that could not be written (vvvvvvvvvv).

Standard Corrective Action: Execution continues, but the record has not been written.

Programmer Response: Change the logic of your program or the order of the records being loaded so that the records are loaded in increasing sequence of their primary key values. Be sure that the key of reference is actually the file's primary key.

IFY141I VCOM2 : RESIDENCY ABOVE 16 MB NOT POSSIBLE RUNNING IN LINK MODE.

Explanation: You are running your program in link mode, and your program resides at an address greater than 16 MB in an MVS/XA system. Execution is impossible in this case since several library routines must run at an address below 16 MB.

Supplemental Data Provided None.

Standard Corrective Action Execution terminates with a return code of 16.

Programmer Response: Either:

1. Do not supply the library SYS1.VLNKMLIB (or the equivalent at your installation) in the SYSLIB DD statement in the linkage editor step when link-editing your program for execution in load mode, or

2. When executing in link-mode, be sure that your load module is given an RMODE value of 24 when it is link-edited. You probably specified an RMODE of ANY; either remove this linkage editor parameter or specify an RMODE value of 24.

IFY142I DCOM2 | VCOM2 : IFYVLBCM IS AT LEVEL lbcm-lvl BUT mod-name IS AT LEVEL mod-lvl.

Explanation: You were running your program in load mode which required loading the composite module *mod-name*. However, the module loaded was from a different release level of the VS FORTRAN Library than the rest of the executing library routines (in particular, different from the release level of the composite module IFYVLBCM). If you are running under CMS, the composite module *mod-name* may be in a discontiguous shared segment.

Supplemental Data Provided: The name of the loaded composite module (*mod-name*), its level (*mod-lvl*), and the level of the executing version of IFYVLBCM (*lbcm-lvl*). The levels are in the form vvrrmm, where vv is the version number, rr is the release number, and mm is the modification number.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Be sure that you are specifying for execution the correct libraries which contain the VS FORTRAN Library. Specify:

- A JOBLIB or STEPLIB DD statement in MVS,
- A GLOBAL LOADLIB command in CMS, or
- DLBL, EXTENT, ASSGN, and LIBDEF statements in VSE.

In addition, be sure that any sharable copies of the composite module are at the same level as the rest of the VS FORTRAN Library you are using for execution. These sharable copies are in a

- Link pack area in MVS,
- Discontiguous shared segment in CMS, or
- Shared virtual area (SVA) in VSE.

For further assistance, refer the problem to the people at your installation who give system support for VS FORTRAN.

IFY143I DGMFM : NO MORE VIRTUAL STORAGE IS AVAILABLE IN THE GETVIS AREA.
IFY143I DGMFM : ERROR RETURN CODE nn WHILE OBTAINING VIRTUAL STORAGE.

Explanation: In attempting to obtain virtual storage in VSE, a library routine was unable to obtain the required storage.

Supplemental Data Provided: In the second form of the message, the return code (nn) returned by the GETVIS macro instruction. (In the first form of the message, the return code was 12.) GETVIS macro code is explained in *VSE/Advanced Functions Macro Reference*.

Standard Corrective Action: The program is cancelled.

Programmer Response: For the first form of the message, provide a SIZE parameter on your EXEC statement to allow a larger a GETVIS area. You may also have to run your program in a larger partition. For the second form of the message, refer the problem to the people at your installation who give system support for VS FORTRAN.

IFY144I DLCI0 | DLOAD : INSUFFICIENT GETVIS SPACE FOR LOADING PHASE phase-name.
IFY144I DLCI0 | DLOAD : ERROR RETURN CODE nn OCCURRED LOADING PHASE phase-name.

Explanation: In attempting to load a phase in VSE, a library routine was unable to load the required phase.

Supplemental Data Provided: In the both forms of the message, the name of the phase (phase-name). In the second form of the message, the return code (nn)

returned by the CDLOAD macro instruction. (In the first form of the message, the return code was 12.) CDLOAD macro code is explained in *VSE/Advanced Function Macro Reference.*

Standard Corrective Action: The program is cancelled.

Programmer Response: For the first form of the message, provide a SIZE parameter on your EXEC statement which allows a larger GETVIS area. You may also have to run your program in a larger partition. For the second form of the message, refer the problem to the people at your installation who give system support for VS FORTRAN.

IFY145I DCOM2 | VCOM2 : COMPOSITE MODULE mod-name IS NOT IN THE EXPECTED FORMAT.

Explanation: You were running your program in load mode which required loading the composite module *mod-name.* However, the module loaded was not recognized as a valid composite module. If you are running under CMS, this composite module may be in a discontiguous shared segment which 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 Library with:

- A JOBLIB or STEPLIB DD statement in MVS,
- A GLOBAL LOADLIB command in CMS, or
- DLBL, EXTENT, ASSGN, and LIBDEF statements in VSE.

Be sure that the composite module has been built properly. Building composite modules is explained in *VS FORTRAN Compiler and Library Installation and Customization.*

If you are executing under CMS and the system name of a discontiguous shared segment has been defined, be sure the shared segment has been built properly.

For further assistance, refer the problem to the people at your installation who give system support for VS FORTRAN.

IFY146I VLINP : THE REENTRANT LOAD MODULE module-name WAS LOADED ABOVE THE 16MB VIRTUAL STORAGE LINE BY PROGRAM program-name, WHICH WAS RUNNING IN 24-BIT ADDRESSING MODE.

Explanation: The reentrant load module contains the program's reentrant CSECT, but because of the module's location and the program's addressing mode, the program can never branch to the CSECT. 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 reentrant load module with an AMODE value of 24.

IFY147I VLINP : THE REENTRANT LOAD MODULE module-name LOADED BY PROGRAM program-name HAS AN INCORRECT FORMAT.

Explanation: A program's nonreentrant CSECT loaded a load module containing the program's reentrant CSECT. The load module, however, was not in the correct format because the CSECTs 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 reentrant and nonreentrant CSECTs of the program. Then link-edit the reentrant CSECT to create the load module.

IFY148I VLINP : THE REENTRANT LOAD MODULE module-name LOADED BY PROGRAM program-name DOES NOT CONTAIN CSECT csect-name AT AN ACCESSIBLE LOCATION.

Explanation: A program's nonreentrant CSECT loaded a load module that does not contain the program's reentrant CSECT.

Supplemental Data Provided: The names of the load module, the program, and the CSECT.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Link-edit the reentrant CSECT (produced by the object-deck separation tool) into the load module.

IFY149I VLINP : THE REENTRANT LOAD MODULE module-name LOADED BY PROGRAM program-name HAS A TIMESTAMP IN CSECT csect-name1 WHICH DIFFERS FROM THAT IN CSECT csect-name2. THE csect-name1 TIMESTAMP IS xxxxxxxxxxxxxx AND THE csect-name2 TIMESTAMP IS yyyyyyyyyyyyyy.

Explanation: A program's nonreentrant CSECT loaded a load module containing the program's reentrant CSECT, but the timestamps of the CSECTs were found to be different. The CSECTs 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 CSECTs, and the timestamps (xxxxxxxxxxxxxx, yyyyyyyyyyyyyy) of the CSECTs.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Make the load module containing the reentrant CSECT available at execution time. Tell your system programmer that the shared segment (in VM) or the link pack area (in MVS) may have to be updated.

IFY150I VLOAD : THE LOAD LIST, WHICH HAS nn ENTRIES, IS FULL. A TOTAL OF mm LOADED MODULES WAS NOT ENTERED INTO THE LIST. SUCH MODULES ARE NOT DELETED.

Explanation: While one module was loading another, the load list was found to be full. Consequently, the name and address of the loaded module cannot be added to the list.

Supplemental Data Provided: The number of entries (nn) in the load list and the number of modules (mm) whose names could not be entered into the load list.

Standard Corrective Action: Execution continues normally, but the loaded module will not be deleted when the program terminates.

Programmer Response: None.

IFY151I VDIOS : nnnn RECORDS OF LENGTH llll FORMATTED ON FILE fffffff.

Explanation: The message tells you 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 file for the first time.

Supplemental Data Provided:

nnnn is the number of records formatted on the file.

llll is the length of the records.

fffffff is the name of the file.

Standard Corrective Action: None.

Programmer Response: None.

IFY152I VSIOS | VDIOS : FILE IS UNUSABLE, PERMANENT ERROR HAS BEEN DETECTED. FILE ffffff

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: ffffff is the name of the unusable file.

Standard Corrective Action: The interrupted instruction and the I/O request are ignored. After the traceback is completed, control is returned to the call routine statement designated in the ERR parameter of an I/O statement, if that parameter 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.

IFY153I VCOM2 : THE PARAMETER LIST RECEIVED FROM rrrrrr IS INCONSISTENT WITH THE ARGUMENTS EXPECTED BY ssssss. INHERITED LENGTH OF A CHARACTER ARGUMENT IS REQUIRED. EXECUTION IS TERMINATED. RECOMPILE TO PREVENT THIS MESSAGE.

Explanation: A dummy argument or a function name is a character entity with inherited length, and there is no secondary list. A subroutine or function (ssssss) was compiled at Release 3.0 or later of VS FORTRAN, and the calling routine (rrrrrr) was compiled at Release 2.0 of VS FORTRAN or a previous release.

Supplemental Data Provided: The name of the main routine and the called subprogram.

Standard Corrective Action: The job will be terminated.

Programmer Response: Recompile routine with Release 3.0 or later.

IFY154I VCOM2 : THE PARAMETER LIST RECEIVED FROM rrrrrr IS INCONSISTENT WITH THE ARGUMENTS EXPECTED BY ssssss. THE PARAMETER LIST IS ACCEPTED BUT EXECUTION RESULTS MAY BE INVALID. RECOMPILE TO PREVENT THIS MESSAGE.

Explanation: The calling routine rrrrrr (or the subprogram ssssss) has been compiled with VS FORTRAN at Release 1.0, 1.1, or 2.0, and the subprogram ssssss (or the calling routine rrrrrr) has been compiled with VS FORTRAN Release 3.0.

Supplemental Data Provided: The name of the calling routine and the called subprogram.

Standard Corrective Action: Execution continues with the first statement in the subprogram.

Programmer Response: Recompile all subprograms with character arguments using VS FORTRAN Release 3.0. For assembler subprograms with character arguments, see Appendix A in *VS FORTRAN Programming Guide*.

IFY155I VOPEN : RECL PARAMETER IS NOT ALLOWED FOR AN OPEN OF A SEQUENTIAL FILE, UNIT nn
IFY155I VOPEN : RECL PARAMETER IS REQUIRED FOR AN OPEN OF A DIRECT ACCESS FILE, UNIT nn

Explanation: For the first form of the message, the RECL= parameter is specified for a sequential file. With the second form of the message, the RECL= parameter was not specified for a direct file.

Supplemental Data Provided: nn is the number of the of the unit specified on the OPEN statement.

Standard Corrective Action: The IOSTAT= variable is set positive and/or the ERR= exit is taken. If neither the IOSTAT= nor ERR= parameter is specified, the program is terminated.

Programmer Response: Correct the program to specify the correct combination of the ACCESS= and RECL= parameters.

IFY156I DYCMN : UNABLE TO OBTAIN STORAGE FOR COMMON 'common-name'.

Explanation: There is insufficient storage available to allow allocation for the named common.

Supplemental Data Provided: The name of the common.

Standard Corrective Action: The request is ignored. Processing continues. Any reference to variables in this common will result in termination of this job.

Programmer Response: Rerun the program with larger storage or recompile the program with smaller common.

IFY157I DYCMP : DYNAMIC COMMON TABLE TABLE FULL. COMMON 'common-name' NOT PROCESSED.

Explanation: There are more than 40 dynamic commons specified in this job.

Supplemental Data Provided: The name of the common.

Standard Corrective Action: Processing is not performed for the specified common. Any reference to variables in this common will result in termination of this job.

Programmer Response: Recompile the job with a smaller number of dynamic commons.

IFY158I DDCMP : LENGTHS OF COMMON 'common-name' ARE NOT CONSISTENT IN ALL MODULES OF THIS PROGRAM

Explanation: A dynamic common must have the same length in all segments of a program.

Supplemental Data Provided: Name of the common.

Standard Corrective Action: Invocation of a subprogram containing a dynamic common whose length differs from that defined in the calling program will result in termination of this job.

Programmer Response: Specify length of the common to be the same in all segments.

IFY159I BTSHS : SECOND ARGUMENT TO function-name FUNCTION IS INVALID.

Explanation: The second argument is not in the valid range for this bit function.

Supplemental Data Provided: The name of the bit function.

Standard Corrective Action: For ISHFT, the result = 0; for IBSET and IBCLR, the result is the first operand; for BTEST, the result is false.

Programmer Response: Specify second argument within allowable range.

IFY160I VCOMH : FORMAT NESTED PARENTHESES TABLE OVERFLOW. REDUCE NUMBER OF NESTED PARENTHESES IN PROGRAM AND RECOMPILE.

Explanation: The format contains more nested parentheses than the library table can hold.

Supplemental Data Provided: None.

Standard Corrective Action: Parenthesis group is ignored. Processing continues. Results are unpredictable.

Programmer Response: Reduce the number of parenthesis groups to 50 or less.

IFY161I VASYP : ASYNCHRONOUS I/O NOT SUPPORTED ON THIS OPERATING SYSTEM (DOS OR CMS)

Explanation: A program called the asynchronous I/O scheduling routine while running in a DOS or CMS environment.

Supplemental Data Provided: TRACEBACK PATH is provided. If GOSTMT is used as a compiler option, TRACEBACK provides the ISN of the I/O statement making the asynchronous I/O request.

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 a MVS system or rewrite the program to use synchronous I/O (unformatted).

IFY162I VVIOS | CVIOS | DVIOS : WRITE STATEMENT CANNOT BE ISSUED TO SEQUENTIALLY ACCESSED VSAM RRDS FILE fffffff

Explanation: An attempt was made to add a record to a sequentially accessed VSAM relative record file that was not empty when the file was opened.

Supplemental Data Provided: The name of the file (fffffff) upon which the request was made.

Standard Corrective Action: The execution is terminated.

Programmer Response: If a record must be added to a nonempty VSAM relative record file, use the access mode of DIRECT.

IFY163I VVIOS | CVIOS | DVIOS : FILE POSITIONING I/O STATEMENT IS NOT ALLOWED IN THE DIRECT ACCESS MODE

Explanation: A file positioning input/output statement (REWIND, BACKSPACE, or ENDFILE) was issued to a VSAM direct file.

Supplemental Data Provided: None.

Standard Corrective Action: The execution is terminated.

Programmer Response: Correct the program so that no file positioning input/output statements are issued for VSAM direct files.

IFY164I VVIOS | CVIOS | DVIOS : RECORD LENGTH OF FILE ffffff 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 parameter when the VSAM cluster is defined using access method services) is less than the length of the record to be written.

Supplemental Data Provided: The name of the file (fffffff) upon which the request was made.

Standard Corrective Action: The execution is terminated.

Programmer Response: Either correct the program so that the length of the record to be written is not greater than the one in the VSAM catalog, or change the record length in the VSAM catalog by redefining the cluster.

IFY165I VVIOS | CVIOS | DVIOS : FILE ffffff IS A KEY SEQUENCED DATA SET WHICH IS BEING OPENED FOR aaaaaaaaaa ACCESS.

Explanation: An attempt was made to open a VSAM KSDS file for sequential or direct access. In such cases, only VSAM ESDS and RRDS files are supported.

Supplemental Data Provided:

fffffff is the name of the file for which the OPEN statement was issued.

aaaaaaaaaa is either SEQUENTIAL or DIRECT.

Standard Corrective Action: The execution is terminated.

Programmer Response: If you want to process the VSAM KSDS, code ACCESS='KEYED' on the OPEN statement. Otherwise, change the data definition statement to refer to a file other than a VSAM KSDS.

IFY166I VVIOS | CVIOS | DVIOS : ENDFILE STATEMENT IS TREATED AS DOCUMENTATION FOR VSAM FILE ffffff

Explanation: A request was made to write an end-of-file mark on a VSAM or VSAM-managed sequential file.

Supplemental Data Provided: The name of the file (ffffff) for which the request was made.

Standard Corrective Action: The request is ignored.

Programmer Response: Remove the statement after carefully checking the effect of removing the statement.

IFY167I VVIOS | CVIOS | DVIOS : ERROR ON VSAM FILE: ffffff WHEN ATTEMPTING TO PROCESS A(N) xxxxxxxxxx OPERATION RC=yy ERROR CODE=zzz

Explanation: An error was detected by VSAM while an input or output statement, indicated by xxxxxxxxxx, was being processed. The return code and the error code returned by VSAM were yy and zzz, respectively.

Supplemental Data Provided: The name of the operation that caused the error and the return and error codes from VSAM. ffffff is the name of the file.

Standard Corrective Action: The execution is terminated.

Programmer Response: Determine the cause of the error by examining the VSAM return and error codes.

IFY168I VVIOS | CVIOS | DVIOS : xxxxxxxxxx IS ISSUED TO UNOPENED VSAM FILE ON UNIT nn

Explanation: An input or output request was made to an unopened VSAM file.

Supplemental Data Provided: The name of the operation (xxxxxxxxxx) issued to an unopened file, and the number of the unit (nn).

Standard Corrective Action: The execution is terminated.

Programmer Response: Make sure that the OPEN statement for the file was successfully executed.

IFY169I DFNTH : EXTENDED PRECISION OPERATION NOT SUPPORTED IN DOS ENVIRONMENT PSW (xxxxxxxxxxxxxxxx).

Explanation: An extended-precision machine operation that is not supported by the machine instruction set was attempted in the DOS/VSE environment. This is generally a divide operation.

Supplemental Data Provided:

In the first form of the message, the Program Status Word (xxxxxxxxxxxxxxxx) at the point of interrupt is given. An IFY210I message with TRACEBACK or a dump of storage follows.

Standard Corrective Action: None.

Programmer Response: Change the program to exclude the unsupported instruction.

IFY170I VDIOS | VSIOS : OPEN OR CLOSE STATEMENT NOT ALLOWED ON OBJECT ERROR UNIT, REQUESTED FILE ffffff

Explanation: An OPEN or CLOSE statement was directed to the unit upon which execution-time error messages are being directed.

Supplemental Data Provided: The name of the file (ffffff) connected to the error message unit.

Standard Corrective Action: The request is ignored and the job terminated if an ERR= or the IOSTAT parameter was not specified in the OPEN or CLOSE.

Programmer Response: Change the program to request I/O to a unit not being used for error messages.

IFY171I VDIOS | VSIOS : CLOSE STATUS OF KEEP IS NOT ALLOWED ON FILE OPENED WITH SCRATCH STATUS, FILE ffffff

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 parameter to agree with the file usage.

IFY172I VDIOS | VSIOS : FILE ffffff 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 OPEN request is ignored.

Programmer Response: Change the program to specify a different unit in the OPEN request or change the logic to use the current unit to which the file is connected.

IFY173I VDIOS | VSIOS : I/O STATEMENT SPECIFYING UNFORMATTED I/O ATTEMPTED ON FORMATTED FILE fffffff

Explanation: FORMATTED and UNFORMATTED I/O requests on the same file are not allowed.

Supplemental Data Provided: The name of the file (fffffff) for which the request was made.

Standard Corrective Action: The I/O operation is ignored.

Programmer Response: Correct the program to direct FORMATTED and UNFORMATTED I/O to different files.

IFY174I VDIOS | VSIOS : I/O STATEMENT SPECIFYING FORMATTED I/O ATTEMPTED ON UNFORMATTED FILE fffffff

Explanation: FORMATTED and UNFORMATTED I/O requests on the same file are not allowed.

Supplemental Data Provided: The name of the file (fffffff) for which the request was made.

Standard Corrective Action: The I/O operation is ignored.

Programmer Response: Correct the program to direct FORMATTED and UNFORMATTED I/O to different files.

IFY175I OPSYS : AN INVALID LITERAL PARAMETER WAS DETECTED IN THE CALL OPSYS STATEMENT.

Explanation: The first parameter in the call to OPSYS did not specify a literal of FILEOPT or LOAD.

Supplemental Data Provided: None.

Standard Corrective Action: The request is ignored.

Programmer Response: Correct the program to specify the correct parameter value.

IFY176I OPSYS : THE FORTRAN LOGICAL UNIT NUMBER IS ASSIGNED TO SYSTEM UNIT, UNIT nn.

Explanation: The unit specified in the call to OPSYS currently has a file connected and cannot be modified.

Supplemental Data Provided: The unit number (nn) specified in the call to OPSYS.

Standard Corrective Action: The request is ignored.

Programmer Response: Correct the program to process the I/O on a different unit.

IFY177I OPSYS : INVALID BLOCK SIZE SPECIFIED; ASCII (18-2048) OR EBCDIC (18-32767), UNIT nn.

Explanation: An invalid block size was specified for the unit set up for ISCII/ASCII or EBCDIC processing.

Supplemental Data Provided: The unit number (nn) specified in the call to OPSYS.

Standard Corrective Action: The request is ignored.

Programmer Response: Correct the program to specify a block size consistent with the file usage.

IFY178I OPSYS : INVALID BUFFER OFFSET SPECIFIED; GREATER THAN 99, EXCEEDS BLOCK SIZE OR IS NEGATIVE, UNIT nn.

Explanation: The buffer offset specified was larger than the block size for the file, or was a negative value, or was a value greater than 99.

Supplemental Data Provided: The unit number (nn) specified in the call to OPSYS.

Standard Corrective Action: The request is ignored.

Programmer Response: Correct the program to specify an offset consistent to the restrictions.

IFY179I OPSYS : AN I/O OPERATION HAS ALREADY BEEN PERFORMED ON THE UNIT, REQUEST IGNORED FOR UNIT nn.

Explanation: An attempt was made to modify the parameters for a file already being used for I/O operations.

Supplemental Data Provided: The unit number (nn) specified in the call to OPSYS.

Standard Corrective Action: The request is ignored.

Programmer Response: Correct the program to process the I/O on a different unit.

IFY180I VINQR | VOPEN : FILE PARAMETER IS NOT VALID FOR AN OPEN STATEMENT, UNIT nn.

Explanation: The FILE parameter on the OPEN statement did not specify a name of 7 characters or less and/or specified a name that did not start with an alphabetic character.

Supplemental Data Provided: The unit number (nn) for which the OPEN statement was issued.

Standard Corrective Action: The OPEN statement is ignored.

Programmer Response: Correct the program to specify a correct file name.

IFY181I VOPEN : STATUS PARAMETER IS NOT VALID FOR AN OPEN STATEMENT, UNIT nn.

Explanation: The STATUS parameter did not specify NEW, OLD, SCRATCH, or UNKNOWN as the status of the file being opened on the unit.

Supplemental Data Provided: The unit number (nn) for which the command was issued.

Standard Corrective Action: STATUS is set to UNKNOWN, and processing continues.

Programmer Response: Correct the program to specify a correct STATUS parameter.

IFY182I VOPEN : ACCESS PARAMETER IS NOT VALID FOR AN OPEN STATEMENT, UNIT nn.

Explanation: The ACCESS parameter did not specify SEQUENTIAL or DIRECT for the type of file access to be employed on the unit.

Supplemental Data Provided: The unit number (nn) for which the OPEN statement was issued.

Standard Corrective Action: The OPEN request is ignored.

Programmer Response: Correct the program to specify a correct ACCESS parameter.

IFY183I VOPEN : BLANK PARAMETER IS NOT VALID FOR AN OPEN STATEMENT, UNIT nn.

Explanation: The BLANK parameter did not specify ZERO or NULL for the treatment of blanks on a FORMATTED I/O request.

Supplemental Data Provided: The unit number (nn) for which the OPEN statement was issued.

Standard Corrective Action: The BLANK parameter is assigned the value NULL.

Programmer Response: Correct the program to specify a correct BLANK parameter.

IFY184I VOPEN : FORM PARAMETER IS NOT VALID FOR AN OPEN STATEMENT, UNIT nn.

Explanation: The FORM parameter did not specify FORMATTED or UNFORMATTED for the file.

Supplemental Data Provided: The unit number (nn) for which the OPEN statement was issued.

Standard Corrective Action: The OPEN request is ignored.

Programmer Response: Correct the program to specify the correct formatting technique.

IFY185I VOPEN : STATUS OF SCRATCH NOT ALLOWED FOR A NAMED FILE OPEN STATEMENT, FILE fffffff.

Explanation: An OPEN requested FILE= and STATUS='SCRATCH' at the same time. The STATUS value is not allowed.

Supplemental Data Provided: The name of file (fffffff) for which the request was made.

Standard Corrective Action: The STATUS value is set to UNKNOWN and processing continues.

Programmer Response: Correct the program to make the two parameters consistent with each other.

IFY186I VCLOS : STATUS PARAMETER IS NOT VALID FOR A CLOSE STATEMENT, UNIT nn.

Explanation: The STATUS parameter 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.

IFY187I VSPAP : (program-name) CALLED SUBROUTINE (program-name) WITH AN ARRAY (array-name (l:u,...)) HAVING LOWER BOUND(S) GREATER THAN UPPER BOUND(S).

Explanation: When one program unit called another, the called program unit was found to have an array with at least one dimension with a lower bound greater than the upper bound.

Supplemental Data Provided: The names of the calling and called program units, the name of the array, and the lower (l) and upper (u) bound of each dimension in the array.

Standard Corrective Action: Execution continues, but invalid results are probable if a reference is made to the dimension(s) in error.

Programmer Response: Correct the specification of dimensions whose lower bound is greater than the upper bound.

IFY188I CITFN : ARGUMENT TO CHARACTER FUNCTION GREATER THAN 255 OR LESS THAN 0.

Explanation: A value greater than 255 (highest EBCDIC representation) or a value less than 0 has been specified for the CHAR function.

Supplemental Data Provided: None.

Standard Corrective Action: The function is not evaluated, and execution continues. The value of the character function is unpredictable.

Programmer Response: Specify correct value.

IFY189I INDEX : INVALID LENGTH FOR INDEX OPERAND lll, VALUE = vvv; VALUE SHOULD BE BETWEEN 1 AND 32767

Explanation: The length specified for the second operand of the index function is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: The length (lll) specified for the operand and its value (vvv).

Standard Corrective Action: The function is not evaluated, and execution continues.

Programmer Response: Specify the correct length.

IFY191I LXCMP : INVALID LENGTH FOR LEXICAL COMPARE, OPERAND xxx. LENGTH VALUE IS: lll.

Explanation: The length specified for the second operand of the LGE, LGT, LLE, or LLT function is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: The operand (xxx) and its length (lll).

Standard Corrective Action: The function is not performed, and execution continues.

Programmer Response: Specify the correct length.

IFY193I CCMPR : INVALID LENGTH FOR CHARACTER COMPARE, OPERAND xxx. LENGTH VALUE IS: lll.

Explanation: The length of the second operand of a character relational compare (.EQ., .LT., ...) not in the range 1 to 32767, inclusive.

Supplemental Data Provided: The operand (xxx) and its length (lll).

Standard Corrective Action: The function is not performed, and execution continues.

Programmer Response: Specify the correct length.

IFY194I VASYP : ASYNCHRONOUS I/O DDNAME "ffffffff", IS LINKED TO A NON-DASD DEVICE

Explanation: The ddname used for asynchronous I/O was determined to be connected to an unusable device type. The only acceptable device types are disk and tape. Terminals, SYSIN, SYSOUT, etc., files are not acceptable.

Supplemental Data Provided: The ddname (ffffffff) of the file on which asynchronous I/O was to be attempted.

Standard Corrective Action: Execution of the program terminates with a return code of 16.

Programmer Response: Connect the file used for asynchronous I/O to an acceptable device type.

IFY195I CMOVE : CHARACTER MOVE INVALID - TARGET AND SOURCE OVERLAP DESTRUCTIVELY.

Explanation: The storage locations assigned to the target and source are such that source data will be destroyed by the requested assignment.

Supplemental Data Provided: None.

Standard Corrective Action: The assignment is not performed, and execution continues.

Programmer Response: Check storage MAP for storage assignments. Also check EQUIVALENCE statements.

IFY196I CMOVE : TARGET LENGTH FOR CHARACTER MOVE GREATER THAN 32767 OR LESS THAN 0.

Explanation: The length of the target (left of equal variable) is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: None.

Standard Corrective Action: The assignment is not performed, and execution continues.

Programmer Response: Specify the correct length.

IFY197I CMOVE : SOURCE LENGTH FOR CHARACTER MOVE GREATER THAN 32767 OR LESS THAN 0.

Explanation: The length of the source (right of equal expression) is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: None.

Standard Corrective Action: The assignment is not performed, and execution continues.

Programmer Response: Specify the correct length.

IFY199I CNCAT : LENGTH FOR CONCATENATION OPERAND GREATER THAN 32767 OR LESS THAN 0.

Explanation: The length of one of the operands of a concatenation operation is not in the range 1 to 32767, inclusive.

Supplemental Data Provided: None.

Standard Corrective Action: The concatenation operation is not performed.

Programmer Response: Specify the correct length.

IFY200I VIIOS : END OF INTERNAL FILE, PROCESSING ENDS.

Explanation: The end of an internal file was reached before the completion of an internal I/O request.

Supplemental Data Provided: None.

Standard Corrective Action: Return to END= label if the request is a READ; otherwise, the job is terminated.

Programmer Response: Either keep a counter to avoid exceeding the end of the internal record or file, or insert an END=n parameter on the READ statement for appropriate transfer of control on end of data.

IFY201I VIOUP : UNFORMATTED VARIABLE SPANNED RECORD IS LONGER THAN THE RECORD LENGTH OF lrecl. THE FILE IS NOT COMPATIBLE WITH NON-FORTRAN ACCESS METHODS. FILE fffffff.

IFY201I VIOUP : UNFORMATTED DIRECT ACCESS DATA IS LONGER THAN THE RECORD LENGTH OF lrecl. THE REMAINING DATA IS TRANSFERRED FROM | TO THE NEXT RECORD. FILE fffffff.

Explanation: Your I/O list items represent a record which is 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 which is longer that the logical record length (LRECL value). For the second format of the message, you are reading or writing from a direct access file and are specifying more data than can be contained in the fixed length records in the file.

Supplemental Data Provided: The record length (lrecl) which is defined for the records in the file and the file name fffffff.

Standard Corrective Procedure: For the first format of the message, a record of the size indicated by your I/O list is written, even though this length exceeds the length defined for the records in the file. If you attempt to read this file using non-FORTRAN access methods, you may encounter unexpected results. For the second format of the message, the next higher numbered record in your direct access file is used to complete the data transfer to or from the items in your I/O list, even though this is in violation of the current FORTRAN standard. For either

format of this message, execution then continues with no further indication that an error occurred.

Programmer Response: To prevent this message from being printed, increase the record length of your file so it is large enough to hold all of 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.

IFY202I VCIA4 : PROGRAM CANNOT BE DEBUGGED WITH RELEASE 1 LEVEL OF IAD.

Explanation: You specified DEBUG as an execution-time parameter which causes the VS FORTRAN Interactive Debug program product to be invoked. However, that program product was found to be at the Release 1 level which is not compatible with the current release of the VS FORTRAN Library.

Supplemental Data Provided: None.

Standard Corrective Action: Execution terminates with a return code of 16.

Programmer Response: Remove the DEBUG parameter from your execution-time parameters so that you will not invoke the VS FORTRAN Interactive Debug program product. You can then run your program without it. Otherwise, refer the problem to the people at your installation who give system support for VS FORTRAN.

IFY203I IBCOP : INVALID COMBINATION OF INITIAL, TEST, AND INCREMENT VALUE FOR READ/WRITE IMPLIED-DO, FILE ffffff; INIT = xxx, = 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:

1. I3=0, or

2. I2 < I1 and I3 ≤ I2-I1, or

3. I1 < I2 and I3 < 0.

Supplemental Data Provided:

ffffff is the name of the file used in the READ or WRITE operation.

xxx is the initial value, yyy the test value, and zzz the 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.

IFY204I VIOLP : ITEM SIZE EXCEEDS BUFFER LENGTH, FILE ffffff

Explanation: For a noncomplex 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.

IFY205I VASYP : I/O SUBTASK ABENDED

Explanation: The asynchronous I/O subtask resulted in an abnormal termination.

Supplemental Data Provided: None.

Standard Corrective Action: Processing is terminated.

Programmer Response: Verify that all DD statements are coded correctly and refer to the appropriate data sets. Check all READ and WRITE statements and any END FILE, REWIND, and BACKSPACE statements. Check the system completion code for assistance in determining the type of error that caused abnormal termination. Increase storage size as a possible solution.

IFY206I VCVTH : INTEGER VALUE OUT OF RANGE (nnnnnnnn)

Explanation: An input integer was too large to fit into the integer data item. (The largest integer that can be processed is 2**15-1 for INTEGER*2 and 2**31-1 for INTEGER*4.)

Supplemental Data Provided: The input integer (nnnnnnnn).

Standard Corrective Action: The maximum positive or negative value will be returned for the size (2 or 4 bytes) of the receiving field.

Programmer Response: Make sure that all integer input data used is within the required range for the integer variable size.

IFY207I

Explanation: Refer to "Program-Interrupt Messages" on page 465 for information on this message.

IFY208I

Explanation: Refer to "Program-Interrupt Messages" on page 465 for information on this message.

IFY209I

Explanation: Refer to "Program-Interrupt Messages" on page 465 for information on this message.

IFY210I

Explanation: Refer to "Program-Interrupt Messages" on page 465 for information on this message.

IFY211I VCOMH : ILLEGAL field FORMAT CHARACTER SPECIFIED (character), FILE fffffff

Explanation: An invalid character has been detected in a FORMAT statement.

Supplemental Data Provided: The field containing the character in error, the character specified, and the file name (fffffff).

Standard Corrective Action: Format field treated as an end of format.

Programmer Response: Make sure that all object-time format specifications are valid.

IFY212I VCOMH : FORMATTED I/O, END OF RECORD, FILE fffffff

Explanation: An attempt has been made to read or write a record, under FORMAT control, that exceeds the buffer length.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: For a read operation, the remainder of the I/O list is ignored; for a write operation, a new record is started with no control character.

Programmer Response: If the error occurs on input, verify that a FORMAT statement does not define a VS 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.

IFY213I VCOMH | VIOLP | VASYP | VXIOS : rrrr END OF RECORD, FILE fffffff

Explanation: For VCOMH and VASYP: The input list in an I/O statement without a FORMAT specification is larger than the logical record.

Supplemental Data Provided: The name of the file (fffffff) and the operation (rrrr).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Make sure the number of elements in the I/O list matches the number of items in the record.

Explanation: For VIOLP: A VS FORTRAN list-directed READ statement attempted to read more items from a variable spanned logical record than were present in the record. (This message can be issued only when the record format is variable spanned.)

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Make sure that the number of items in the I/O list and the input data agree. Either delete extra variable names or supply additional logical records.

IFY214I DSIOS | VSIOS | VASYP : RECORD FORMAT INVALID FOR UNFORMATTED OR ASYNCHRONOUS I/O, FILE ffffff

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.

IFY215I 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.

IFY216I VSIOS : INVALID USE OF I/O CONTROL COMMAND AT LOAD POINT, FILE ffffff

Explanation: The use of a BACKSPACE control command was recognized when the file was at the start of the first record.

Supplemental Data Provided: The name of the file (ffffff) for which command was issued.

Standard Corrective Action: The control command is ignored.

Programmer Response: Correct the program to ensure that a BACKSPACE will not occur at the first command for a file.

IFY217I name : END OF DATA SET, FILE fffffff.

Explanation: An end-of-data-set was sensed during a READ operation; that is, a program attempted to read beyond the end of the data.

Supplemantal Data Provided: The last 5 characters in the name of the module that issued the message: CVIOS, DSIOS, VSIOS, VASYP, or VVIOS. The name of the file (fffffff).

Standard Corrective Action: The next file is read, that is, the data set sequence number is incremented by 1 in the MVS and VM environments. A permanent I/O error is set for the VSE environment.

Programmer Response: Either keep a counter to avoid exceeding the end of record or file, or insert an END=n parameter on the READ statement for appropriate transfer of control on end of data set. Check all job control statements.

IFY218I name: I/O ERROR, FILE fffffff, cccccccccc ERROR OCCURRED WHILE PROCESSING STATEMENT nnnn.

Explanation: One of the following occurred:

- A permanent I/O error has been encountered.
- For sequential I/O, the length of a physical record is inconsistent with the default block size or the block size specified on the job control statement.
- An attempt has been made to read or write a record that is less than 18 bytes long on magnetic tape.
- End-of-tape was encountered while writing a tape file.
- For VSE only, the program attempted to process multiple files.
- For VM only, the program arrived at the end of the medium.

Note: If a permanent I/O error has been detected while writing in the object error unit data set, the error message is written to the programmer either at the terminal or the SYSOUT data set, and job execution is terminated.

Supplemental Data Provided: The last 5 characters in the name of the module that issued the message: DSIOS, VSIOS, VASYP, or VDIOS. The name of the file (fffffff), a character string (cccccccccc) that specifies the type of I/O error, and the number (nnnn) of the statement label or ISN. The short form gives I/O error, file f, and c. The long form gives I/O error, file f, error occurred, but nnnn is not present.

Standard Corrective Action: The I/O request is ignored. After the traceback is completed, control is returned to the call routine statement designated in the ERR=n parameter of an I/O statement, if that parameter was specified.

Programmer Response: For sequential I/O, make sure that the length of the physical record is consistent with the default or specified block size. Check all job control statements. Make sure that no attempt has been made to read or write a magnetic tape record that is fewer than 18 bytes in length.

IFY219I **name : OPEN FAILED, MISSING OR INVALID CONTROL STATEMENT, FILE fffffff**

Explanation:

FOR EBCDIC DATA SETS: Either a data set is referred to in the load module and no job control statement is supplied for it, or a job control statement has an erroneous file name.

Supplemental Data Provided: The last 5 characters in the name of the module that issued the message: DDIOS, DSIOS, VSIOS, VASYN, or VDIOS. The name of the file (fffffff).

Standard Corrective Action: The OPEN request is ignored and execution continues.

Note: If no job control statement has been supplied for the object error unit data set, the message is written either to the programmer at the terminal or console or to the SYSOUT data set, and the job is terminated.

Programmer Response: Either provide the missing job control statement, or correct any erroneous job control statement. Refer to *VS FORTRAN Programming Guide* for more information.

FOR ISCII/ASCII DATA SETS:

A data set may have been referred to in the load module but had no corresponding job control statement, or the job control statement may have had an erroneous filename.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The I/O request is ignored and execution continues.

Programmer Response: Either provide the missing job control statement, or correct any erroneous filename. Also, for MVS files, be sure that the LABEL parameter on the DD statement specifies AL (or NL, provided that the DCB subparameter OPTCD=Q is also specified). Also, be sure that the operating system permits the use of ASCII data sets.

IFY220I 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 The last five characters in the name of the module that issued the message: VSIOS, VDIOS, DIOCS, DVIOS, CVIOS, DDIOS, DSIOS, VVIOS, or VASYN and the unit number (nn).

Standard Corrective Action: The statement is ignored, and execution continues.

Programmer Response: Correct the invalid unit number.

IFY221I VIONP : NAME LONGER THAN EIGHT CHARACTERS. NAME=name

Explanation: An input variable name is longer than eight characters.

Supplemental Data Provided: The first eight 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.

IFY222I 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.

IFY223I 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.

IFY224I VIONP : SUBSCRIPT FOR NON-DIMENSIONED VARIABLE OR SUBSCRIPT OUT OF RANGE. NAME=name

Explanation: A subscript is encountered after an undimensioned input name, or the subscript is too large.

Supplemental Data Provided: The name of the item.

Standard Corrective Action: The remainder of the NAMELIST request is ignored.

Programmer Response: Insert any missing DIMENSION statements, or correct the invalid array reference.

IFY225I VCVTH : ILLEGAL HEXADECIMAL CHARACTER char

Explanation: An invalid character is encountered on input for the Z format code.

Supplemental Data Provided: A display of the record in which the character appeared.

Standard Corrective Action: 0 replaces the encountered character.

Programmer Response: Either correct the invalid character, or correct or delete the Z format code.

IFY226I 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 process 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.

IFY227I VIOLP : ERROR IN REPEAT COUNT, FILE fffffff

Explanation: An invalid condition was detected while scanning for a (k*---):

- an invalid character was found at the start of the scan,
- a secondary repeat count was detected while under the control of a primary repeat count, or
- the numeric value of the repeat count was invalid.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The remainder of the I/O list is ignored.

Programmer Response: Make sure that all repeat counts are correctly specified.

IFY228I VASYP : LAST ITEM IN THE I/O LIST HAS A LOWER ADDRESS THAN THE FIRST ELEMENT, FILE ffffff

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.

IFY230I 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.

IFY231I VSIOS | DSIOS : SEQUENTIAL I/O ATTEMPTED ON A aaaaaa FILE. UNIT nn where aaaa = direct or keyed

IFY231I 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. in such a case.

Supplemental Data Provided: The unit number (nn).

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

IFY232I 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, DVIOS, or CVIOS. The record number (nnnnnn) and the name of the file (fffffff).

Standard Corrective Action: The I/O request is ignored.

Programmer Response: Make sure that the relative position of the record on the data set has been specified correctly. Check all job control statements.

IFY233I VDIOS : RECORD LENGTH GREATER THAN 32760 SPECIFIED, FILE fffffff

Explanation: The record length specified in the DEFINE FILE or OPEN statement exceeds the capabilities of the system and the physical limitation of the volume assigned to the data set in the job control statement.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The record length is set to 32000.

Programmer Response: Make sure that appropriate parameters of the job control statement conform to specifications in the DEFINE FILE or OPEN statement; the record length in both must be equivalent and within the capabilities of the system and the physical limitations of the assigned volume.

IFY234I 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.

IFY235I VSIOS | DSIOS : DIRECT I/O ATTEMPTED ON A aaaaaaaaaa FILE. UNIT nn.

Explanation: Direct I/O statements were used for a file open for sequential or keyed access. A program unit cannot use direct I/O statements in such a case.

Supplemental Data Provided:

nn is the unit number specified in the I/O statement.

aaaaaaaaaa is either SEQUENTIAL or KEYED.

Standard Corrective Action: The I/O request is ignored.

Programmer Response:

- If you want to do direct I/O processing, statement include the necessary DEFINE FILE or OPEN for direct access.
- Make sure the same file name is not used twice within the same program unit for different types of access.

IFY236I VDIOS : DIRECT ACCESS READ REQUESTED BEFORE FILE WAS CREATED, FILE ffffff

Explanation: A READ is executed for a direct access file that has not been created.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The I/O request is ignored.

Programmer Response: Make sure that either a file utility program has been used, or appropriate parameters have been specified on the associated job control statement. For further information, refer to *VS FORTRAN Application Programming: Guide*.

IFY237I VDIOS : INCORRECT RECORD LENGTH SPECIFIED, FILE fffffff

Explanation: The length of the record did not correspond to the length of the record specified in the DEFINE FILE or the OPEN statement.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The I/O request is ignored.

Programmer Response: Make sure that the length of the records supplied matches the length specified in the DEFINE FILE or the OPEN statement. If necessary, change the statement to specify the correct record length.

IFY238I VIOLP : INCORRECT DELIMITER IN COMPLEX OR LITERAL INPUT, FILE ffffff

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.

IFY239I VASYP : BLKSIZE IS NOT SPECIFIED FOR AN INPUT FILE, FILE ffffff

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.

IFY240I

VSTAE : ABEND CODE IS: SYSTEM sss, USER uuu.
SCB/SDWA=aaaaaaaa
IO ccccccccc. PSW=xxxxxxxxxxxxxxxx ENTRY POINT=eeeeeeee.
REGS 0 - 3 nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn
REGS 4 - 7 nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn
REGS 8 -11 nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn
REGS 12-15 nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn
FRGS 0 & 2 nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn
FRGS 4 & 6 nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn
DYNAMIC COMMON MAP
dddddd AT ffffffff (gggggggg) ddddddd AT ffffffff(gggggggg)
MAP FOR REENTRANT LOAD MODULE: bbbbbbbb
hhhhhhhh AT fffffff hhhhhhhh AT fffffff
LOADED LIBRARY MODULES
jjjjjjjj AT fffffff jjjjjjjj AT ffffffff

Explanation: An abnormal program termination has occurred. Message IFY240I is printed on the object program error unit and included in the message class for the job. The IFY240I message may be preceded by an IFY210I message that is printed on the object program error unit.

Supplemental Data Provided: sss is the completion code whan a system code caused termination; uuu is the completion code when a program code caused termination.

For specific explanations of the completion codes, see the messages and codes manual that applies to your operating system.

The SCB/SDWA field gives the address (aaaaaaaa) of the system diagnostic work area, which contains the old PSW (xxxxxxxxxxxxxxxx) and the contents of the general and floating-point registers at the time of the abend. These fields have been copied from the SDWA into this message.

The status of input/output operations is shown in the field IO ccccccccc. 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 COMMONs have been used, a map of obtained COMMON areas is provided where dddddd is the name of the COMMON, fffffff is the starting address of the COMMON, and gggggggg is the length in hexadecimal. If reentrant FORTRAN routines have been loaded, a map of the reentrant CSECTS is provided where hhhhhhhh is the reentrant CSECT name and fffffff 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 fffffff is the address of the module.

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 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 IFY240I 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. See "Output from Symbolic Dumps" on page 447 for an explanation of the SDUMP output.

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.

IFY241I FIXPI : INTEGER BASE=0, INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO

Explanation: For an exponentiation operation (I**J) in the subprogram IFYFIXPI (FIXPI#), where I and J represent integer variables or integer constants, I is equal to zero and J is less than or equal to zero.

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.

IFY242I FRXPI : REAL*4 BASE=0.0, INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO

Explanation: For an exponentiation operation (R**J) in the subprogram IFYFRXPI (FRXPI#), where R represents a REAL*4 variable or REAL*4 constant and J represents an integer variable or integer constant, R is equal to 0 and J is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=0.0,EXP<0,RESULT=0;
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.

IFY243I FDXPI : REAL*8 BASE=0.0, INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO ZERO

Explanation: For an exponentiation operation (D**J) in the subprogram IFYFDXPI (FDXPI#), where D represents a REAL*8 variable or REAL*8 constant and J represents an Integer variable or Integer constant, D is equal to 0 and J is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action: The• is the correctly signed largest representable floating-point number.

If BASE=0.0,EXP<0,RESULT=•;
If BASE=0.0,EXP=0,RESULT=1.

Programmer Response: Make sure that both the real variable or constant base and the integer variable or constant exponent for an exponentiation operation are within the allowable range. If the base and exponent may or will fall outside that range during execution, then either modify the operand(s), or insert source code to test for the situation and make the appropriate compensation. Bypass the exponentiation operation if necessary.

IFY244I 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 IFYFRXPR (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.

IFY245I 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 IFYFDXPD (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: Result=0.

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.

IFY246I FCXPI : COMPLEX*8 BASE = (0.0,0.0), EXPONENT=exponent LESS THAN OR EQUAL TO 0

Explanation: For an exponentiation operation (Z**J) in the subprograms IFYFCXPI (FCXPI#) and IFYFCXPC (FCXPC#), where Z represents a COMPLEX*8 variable or COMPLEX*8 constant and J represents an integer variable or integer constant, Z is equal to 0 and J is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=0.0,0.0,EXP < 0,RESULT=•+0i;
If BASE=0.0,0.0,EXP=0,RESULT=1+0i

Programmer Response: Make sure that both the complex 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 compensation. Bypass the exponentiation operation if necessary.

IFY247I FCDCD : COMPLEX*16 BASE = (0.0,0.0), EXPONENT=exponent, LESS THAN OR EQUAL TO 0

Explanation: For an exponentiation operation (Z**J) in the subprograms IFYFCDXI (FCDXI#) and IFYFCDCD (FCDCD#), where Z represents a COMPLEX*16 variable or COMPLEX*16 constant and J represents an integer variable or integer constant, Z is equal to zero and J is less than or equal to zero.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=(0.0,0.0)EXP < 0,RESULT=•+0i;
If BASE=(0.0,0.0)EXP=0,RESULT=1+0i

Programmer Response: Make sure that both the complex 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 compensation. Bypass the exponentiation operation if necessary.

IFY248I FQXPI : REAL*16 BASE=0.0, INTEGER EXPONENT=exponent, LESS THAN OR EQUAL TO 0

Explanation: For an exponentiation operation (Q**J) in the subprogram IFYFQXPI (FQXPI#), where Q represents a REAL*16 variable or constant and J represents an integer variable or constant, Q is equal to 0 and J is less than or equal to 0.

Supplemental Data Provided: The exponent specified.

Standard Corrective Action:

If BASE=0.0,EXP < 0,RESULT=•;
If BASE=0.0,EXP=0,RESULT=1

Programmer Response: Make sure that both the real variable or constant base and the integer variable or constant exponent for an exponentiation operation are within the allowable range. If the base and exponent may or will fall outside that range during execution, then either modify the operand(s), or insert source code to test for the situation and make the appropriate compensation. Bypass the exponentiation operation if necessary.

IFY249I 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 IFYFQXPQ (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.

IFY250I 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 IFYFQXPQ, (FQXPQ#) the argument combination of $Y*\log_2(X)$ generates a number greater than or equal to 252.

Supplemental Data Provided: The arguments specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the base and exponent are within the allowable range. If necessary, restructure arithmetic operations.

IFY251I SSQRT : ARG=argument, LESS THAN ZERO

Explanation: In the subprogram IFYSSQRT (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.

IFY252I SEXP : ARG=argument, GREATER THAN 174.673

Explanation: In the subprogram IFYSEXP (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.

IFY253I SLOG : ARG=argument, LESS THAN OR EQUAL TO ZERO

Explanation: In the subprogram IFYSLOG (ALOG and ALOG10), the argument is less than or equal to 0. Because this subprogram is called by an exponential subprogram, this message may also indicate that an attempt has been made to raise a negative base to a real power.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If X=0, RESULT=-•;
If $X < 0$, RESULT=$\log|X|$ or $\log_{10}|X|$.

Programmer Response: Make sure that the argument to the 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.

IFY254I SSCN: ABS(ARGUMENT)=argument GREATER THAN OR EQUAL TO PI*(218)**

Explanation: In the subprogram IFYSSCN (SIN and COS), the absolute value of an argument is greater than or equal to 2**18 * pi (2**18 * pi=.823 550 E+ 06).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=SQRT(2)/2.

Programmer Response: Make sure that the argument (in radians where 1 radian is equivalent to 57.298°) to the trigonometric sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY255I SATN2 : ARGUMENTS = 0.0

Explanation: In the subprogram IFYSATN2, when the entry name ATAN2 is used, both arguments are equal to 0.

Supplemental Data Provided: None.

Standard Corrective Action: Result=0.

Programmer Response: Make sure that both arguments do not become 0 during program execution, or are not inadvertently initialized or modified to 0. Provide code to test for the situation and, if necessary, modify the arguments or bypass the source referencing the function subprogram.

IFY256I SSCNH: ARG=argument, GREATER THAN OR EQUAL TO 175.366

Explanation: In the subprogram IFYSSCNH (SINH or COSH), the argument is greater than or equal to 175.366.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: SINH(X)=±•; COSH(X)=•

Programmer Response: Make sure that the argument to the hyperbolic sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY257I SASCN : ARG=argument, GREATER THAN 1

Explanation: In the subprogram IFYSASCN (ASIN or ACOS), the absolute value of the argument is greater than 1.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If x > 1.0, ACOS(x) = 0;
If x < -1.0, ACOS(x) = pi;
If x > 1.0, ASIN(x) = pi/2;
If x < -1.0, ASIN(x) = -pi/2.

Programmer Response: Make sure that the argument to the arcsine or arccosine function is between -1 and +1, inclusive. If the argument may or will fall outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY258I STNCT : ARG=argument, (HEX=hexadecimal), GREATER THAN OR EQUAL TO PI*(218)**

Explanation: In the subprogram IFYSTNCT (TAN or COTAN), the absolute value of the argument is greater than or equal to 2**18*pi (2**18*pi=.823 550E+6).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=1.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.2958°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY259I STNCT : ARG = argument, (HEX = hexadecimal), APPROACHES SINGULARITY

Explanation: In the subprogram IFYSTNCT (TAN or COTAN), the argument value is too close to one of the singularities (±pi/2, ±3pi/2, ... for the tangent or ±pi, ±2pi, ... for the cotangent).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.2958°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will approach the corresponding singularities for the function during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY260I FQXPR : REAL*16 EXPONENT = exponent, GREATER THAN OR EQUAL TO 252

Explanation: In the subprogram IFYFQXPR (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.

IFY261I LSQRT : ARG = argument, LESS THAN ZERO

Explanation: In the subprogram IFYLSQRT (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.

IFY262I LEXP : ARG = argument, GREATER THAN 174.673

Explanation: In the subprogram IFYLEXP (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.

IFY263I LLOG : = argument, LESS THAN OR EQUAL TO ZERO

Explanation: In the subprogram IFYLLOG (DLOG and DLOG10), the argument is less than or equal to 0. Because the subprogram is called by an exponential subprogram, this message may also indicate that an attempt has been made to raise a negative base to a real power.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If X= 0 ,RESULT=-•;
If X<0,RESULT=log|X| or $\log_{10}$ |X|.

Programmer Response: Make sure that the argument to the 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.

IFY264I LSCN : ABS(ARG) = argument, GREATER THAN OR EQUAL TO PI*(250)**

Explanation: In the subprogram IFYLSCN (DSIN and DCOS), the absolute value of the argument is greater than or equal to 2**50*pi (2**50*pi=.353 711 887 378 022 39D+16).

Supplemental Data Provided: None.

Standard Corrective Action: Result=SQRT(2)/2.

Programmer Response: Make sure that the argument (in radians where 1 radian is equivalent to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the argument may or will exceed that range during

program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY265I LATN2 : ARGUMENTS = 0.0

Explanation: In subprogram IFYLATN2, when entry name DATAN2 is used, both arguments are equal to zero.

Supplemental Data Provided: None.

Standard Corrective Action: Result=0.

Programmer Response: Make sure that both arguments do not become zero during program execution, or are not inadvertently initialized or modified to zero. Provide code to test for the situation and, if necessary, modify the arguments or bypass the source referencing the function subprogram.

IFY266I SCNH : ARG = argument, GREATER THAN OR EQUAL TO 175.366

Explanation: In the subprogram IFYSCNH (DSINH or DCOSH), the absolute value of the argument is greater than or equal to 175.366.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: DSINH(X)=±•; DCOSH(X)=•

Programmer Response: Make sure that the argument to the hyperbolic sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY267I LASCN : ARG = argument, GREATER THAN 1

Explanation: In the subprogram IFYLASCN (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.

IFY268I LTNCT : ARG = argument, (HEX = hexadecimal), GREATER THAN OR EQUAL TO PI*(250)**

Explanation: In the subprogram IFYLTNCT (DTAN or DCOTAN), the absolute value of the argument is greater than or equal to 2**50*pi (2**50*pi=.353 711 887 601 422 01D+16).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=1.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY269I LTNCT : ARG = argument, (HEX = hexadecimal), APPROACHES SINGULARITY

Explanation: In the subprogram IFYLTNCT (DTAN or DCOTAN), the argument value is too close to one of the singularities (±pi/2, ±3pi/2, ... for the tangent; ±pi, ±2pi, ... for the cotangent).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument (in radians where 1 radian is equivalent to 57.295 779 513 1°0) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will approach the corresponding singularities for the function during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY270I FCQXI : COMPLEX*32 BASE = (0.0,0.0), INTEGER EXPONENT = exponent, LESS THAN OR EQUAL TO 0

Explanation: In the subprograms IFYFCQXI (FCQXI#) and IFYFCQCG (FCQCG#), a base 0 number has been raised to an integer power less than or equal to zero.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If X=0+0i and J=0, RESULT=1+0i;
If X=0+0i and J<0, RESULT=•+0i.

(where J=exponent)

Programmer Response: Make sure the base is a nonzero number or change the exponent to a nonzero value.

IFY271I **CSEXP : REAL ARGUMENT = argument, (HEX = hexadecimal), GREATER THAN 174.673**

Explanation: In the subprogram IFYCSEXP (CEXP), the value of the real part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•(COS X + iSIN X) where X is the imaginary portion of the argument.

Programmer Response: Make sure that the argument to the exponential function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY272I **CSEXP : IMAGINARY ARGUMENT = argument, (HEX = hexadecimal), ABSOLUTE VALUE GREATER THAN OR EQUAL TO PI*(2**18)**

Explanation: In the subprogram IFYCSEXP (CEXP), the absolute value of the imaginary part of the argument is greater than or equal to 2**18*pi (2**18*pi=.823 550E+6).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If x is the real part of the argument, then Result=e^x+0*i, where e is the base of natural logarithms.

Programmer Response: Make sure that the argument to the exponential function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation, and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY273I **CSLOG : ARGUMENT = (0.0,0.0)**

Explanation: In the subprogram IFYCSLOG (CLOG), the real and imaginary parts of the argument are equal to zero.

Supplemental Data Provided: None.

Standard Corrective Action: Result=-• +0i.

Programmer Response: Make sure that both the real and imaginary parts of the argument do not become zero during program execution, or are not inadvertently initialized or modified to zero. Provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY274I **CSSCN : REAL ARGUMENT = argument, (HEX = hexadecimal), GREATER THAN OR EQUAL TO PI*(2**18)**

Explanation: In the subprogram IFYCSSCN (CSIN or CCOS), the absolute value of the real part of the argument is greater than or equal to 2**18*pi (2**18*pi=.823 550E+6).

Supplemental Data Provided: The argument specified. The real part is set to zero and the computations are redone.

Standard Corrective Action: The real part is set to zero and the computations are redone. If argument is x + iy, then

CCOS Result=COSH(y)+0*i; CSIN Result=0+SINH(y)*i.

where y is the imaginary part of the original argument.

Programmer Response: Make sure that the real part of the argument (in radians where 1 radian is equivalent to 57.2958°) to the trigonometric sine or cosine function is within the allowable range. If the real part of the argument may or will exceed the range during program execution, then provide code to test for the situation and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

IFY275I **CSSCN : IMAGINARY ARGUMENT = argument, (HEX = hexadecimal), GREATER THAN 174.673**

Explanation: In the subprogram IFYCSSCN (CSIN or CCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If imaginary part > 0 (X is real portion of argument):

For sine, result=•/2*(SIN X + iCOS X).
For cosine, result=•/2*(COS X - iSIN X).

If imaginary part < 0 (X is real portion of argument):

For sine, result=•/2*(SIN X - iCOS X).
or cosine, result=•/2*(COS X + iSIN X).

Programmer Response: Make sure that the imaginary part of the argument (in radians where 1 radian is equivalent to 57.2958°) to the trigonometric sine or cosine function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

IFY276I CQEXP : REAL ARGUMENT = argument, (HEX = hexidecimal), GREATER THAN 174.673

Explanation: In the subprogram IFYCQEXP (CQEXP), the value of the real part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result =•(COS X + iSIN X), where X is the imaginary portion of the argument.

Programmer Response: Make sure that the real part of the argument to the exponential function is within the allowable range. If the real part of the argument may or will exceed the range during program execution, then provide code to test for the situation, and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

IFY277I CQEXP : IMAGINARY ARGUMENT = argument, (HEX = hexadecimal), ABSOLUTE VALUE GREATER THAN PI*(2100)**

Explanation: In the subprogram IFYCQEXP (CQEXP), the absolute value of the imaginary part of the argument is greater than 2**100*pi (2**100*pi=.398 244 181 299 569 74D + 31)

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If x is the real part of the argument, then Result=e^x+0*i, where e is the base of natural logarithms.

Programmer Response: Make sure that the imaginary part of the argument to the exponential function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

IFY278I CQLOG : ARGUMENT = (0.0,0.0)

Explanation: In the subprogram IFYCQLOG (CQLOG), the real and imaginary parts of the argument are equal to 0.

Supplemental Data Provided: None.

Standard Corrective Action: Result=-•+0i.

Programmer Response: Make sure that both the real and imaginary parts of the argument do not become 0 during program execution, or are not inadvertently initialized or modified to 0. Provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY279I **CQSCN : REAL ARGUMENT = argument, GREATER THAN OR EQUAL TO 2**100**

Explanation: In the subprogram IFYCQSCN (CQSIN or CQCOS), the absolute value of the real part of the argument is greater than or equal to 2^{100}.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If the argument is X + iY, for CQSIN, result=0 + DSINH(Y)*i and, for CQCOS, result = DCOSH(Y)+0*i.

Programmer Response: Make sure that the real part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the part of the argument may or will exceed the range during program execution, then provide code to test for the situation and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

IFY280I **CQSCN : IMAGINARY ARGUMENT = argument, GREATER THAN 174.673**

Explanation: In the subprogram IFYCQSCN (CQSIN or CQCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If imaginary part > 0 (X is real portion of argument):

For sine, result=•/2*(SIN X + iCOS X).

For cosine, result=•/2*(COS X - iSIN X).

If imaginary part <0, (X is real portion of argument):

For sine, result=•/2*(SIN X - iCOS X).

For cosine, result=•/2*(COS X + iSIN X).

Programmer Response: Make sure that the imaginary part of the argument (in radians, where 1 radian is equal to 57.2957795131°) to the trigonometric sine or cosine function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

IFY281I **CLEXP : REAL ARGUMENT = argument, (HEX = hexadecimal), GREATER THAN 174.673**

Explanation: In the subprogram IFYCLEXP (CDEXP), the value of the real part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•(COS X + iSIN X) where X is the imaginary portion of the argument.

Programmer Response: Make sure that the real part of the argument to the exponential function is within the allowable range. If the real part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

IFY282I CLEXP : IMAGINARY ARGUMENT = argument, (HEX = hexadecimal), ABSOLUTE VALUE GREATER THAN OR EQUAL TO PI*(250)**

Explanation: In the subprogram IFYCLEXP (CDEXP), the absolute value of the imaginary part of the argument is greater than or equal to 2**50*pi (2**50*pi=.353 711 887 601 422 01D+16).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If X is the real part of the x argument, then Result=e+0*i, where e is the base of natural logarithms.

Programmer Response: Make sure that the imaginary part of the argument to the exponential function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation, and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

IFY283I CLLOG : ARGUMENT = (0.0,0.0)

Explanation: In the subprogram IFYCLLOG (CDLOG), the real and imaginary parts of the argument are equal to 0.

Supplemental Data Provided: None.

Standard Corrective Action: Result=-•+0i.

Programmer Response: Make sure that both the real and imaginary parts of the argument do not become 0 during program execution, or are not inadvertently initialized or modified to 0. Provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY284I CLSCN : REAL ARGUMENT = argument, (HEX = hexadecimal), GREATER THAN OR EQUAL TO PI*(250)**

Explanation: In the subprogram IFYCLSCN (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 = .353 711 887 601 422 01D+16).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If the argument is X + iY, for CDSIN, the result=0 + DSINH(Y) +i; for CDCOS, the result=DCOSH(Y)+0*i.

Programmer Response: Make sure that the real part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the part of the argument may or will exceed the range during program execution, then provide code to test for the situation, and, if necessary, modify the real part of the argument or bypass the source referencing the function subprogram.

IFY285I CLSCN : IMAGINARY ARGUMENT = argument, (HEX = hexadecimal), GREATER THAN 174.673

Explanation: In the subprogram IFYCLSCN (CDSIN or CDCOS), the absolute value of the imaginary part of the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If imaginary part >0, (X is real portion of argument):

For sine, result=•/2*(SIN X + iCOS X).

For cosine, result=•/2*(COS X - iSIN X).

If imaginary part <0, (X is real portion of argument):

For sine, result=•/2*(SIN X - iCOS X).

For cosine, result=•/2*(COS X + iSIN X).

Programmer Response: Make sure that the imaginary part of the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the imaginary part of the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the imaginary part of the argument or bypass the source referencing the function subprogram.

IFY286I VSIOS | VASYP : ATTEMPT TO ISSUE SYNCHRONOUS AND ASYNCHRONOUS I/O REQUESTS WITHOUT AN INTERVENING REWIND, FILE ffffff

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: The I/O request is ignored, and execution continues.

Programmer Response: Insert a REWIND statement at an appropriate point in the program.

IFY287I VASYP : A WAIT ISSUED WITH NO OUTSTANDING I/O REQUEST, FILE fffffff

Explanation: A WAIT statement was issued with no corresponding READ or WRITE request.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: The WAIT statement is ignored, and execution continues.

Programmer Response: Remove the WAIT statement, or include a corresponding READ or WRITE statement.

IFY288I VASYP : NO WAIT ISSUED FOR AN OUTSTANDING I/O REQUEST FILE fffffff

Explanation: No WAIT statement was issued for an outstanding READ or WRITE request.

Supplemental Data Provided: The name of the file (fffffff).

Standard Corrective Action: Execution continues with an implied WAIT.

Programmer Response: Include the WAIT statement, or remove the READ or WRITE statement.

IFY289I QSQRT : NEGATIVE ARGUMENT = argument

Explanation: In the subprogram IFYQSQRT (QSQRT#), the argument is less than zero.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result = $|x|^{1/2}$

Programmer Response: Make sure that the argument is within the allowable range. Either modify the argument, or insert source code to test for a negative argument and make the necessary compensation. Bypass the function reference if necessary.

IFY290I SGAMA : ARG = argument, (HEX = hexadecimal), LESS THAN OR EQUAL TO 2-252 OR GREATER THAN OR EQUAL TO 57.5744**

Explanation: In the subprogram IFYSGAMA (GAMMA), the value of the argument is outside the valid range (2**-252 < x < 57.5744).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the gamma function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY291I SGAMA : ARG = argument, (HEX = hexadecimal), LESS THAN OR EQUAL TO ZERO OR GREATER THAN OR EQUAL TO 4.2937*1073**

Explanation: In the subprogram IFYSGAMA (ALGAMA), the value of the argument is outside the valid range (0 < < 4.2937x10**73).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the ALGAMA function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY292I FQXPR : ARG = argument, GREATER THAN 174.673

Explanation: In the subprogram IFYFQXPR (QEXP), the argument is greater than 174.673.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the exponential function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY293I QLOG : ARG = argument, LESS THAN OR EQUAL TO ZERO

Explanation: In the subprogram IFYQLOG (QLOG and QLOG10), the argument is less than or equal to 0. Because the subprogram is called by an exponential subprogram, this message may also indicate that an attempt has been made to raise a negative base to a real power.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: If X=0, result=-•; if X<0, result=$\log|X|$ or $\log_{10}|X|$.

Programmer Response: Make sure that the argument to the logarithm function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY294I QSCN : ARG = argument, GREATER THAN OR EQUAL TO 2100**

Explanation: In the subprogram IFYQSCN (QSIN and QCOS), the absolute value of the argument is greater than or equal to 2^{100}.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=SQRT(2)/2

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY295I QATN2 : ARGUMENTS = 0.0

Explanation: In subprogram IFYQATN2, when entry name QATAN2 is used, both arguments are equal to zero.

Supplemental Data Provided: None.

Standard Corrective Action: Result=0.

Programmer Response: Make sure that both arguments do not become zero during program execution, or are not inadvertently initialized or modified to zero. Provide code to test for the situation and, if necessary, modify the arguments or bypass the source referencing the function subprogram.

IFY296I QSCNH : ARG = argument, GREATER THAN 175.366

Explanation: In the subprogram IFYQSCNH (QSINH or QCOSH), the absolute value of the argument is greater than (or equal to) 175.366.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: QSINH(X)=±•; QCOSH(X)=•.

Programmer Response: Make sure that the argument to the hyperbolic sine or cosine function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY297I QASCN : ARG = argument, GREATER THAN 1

Explanation: In the subprogram IFYQASCN (QARSIN or QARCOS), the absolute value of the argument is greater than 1.

Supplemental Data Provided: The argument specified.

Standard Corrective Action:

If X < 1.0 QARCOS(X) = 0;
If X < -1.0 QARCOS(X) = pi;
If X > 1.0 QARSIN(X) = pi/2;
If X < -1.0 QARSIN(X) = -pi/2.

Programmer Response: Make sure that the argument to the arcsine or arccosine function is between -1 and +1, inclusive. If the argument may or will fall outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY298I QTNCT : ARG = argument, GREATER THAN OR EQUAL TO 2100**

Explanation: In the subprogram IFYQTNCT (QTAN or QCOTAN), the absolute value of the argument is greater than or equal to 2**100.

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=1.

Programmer Response: Make sure that the argument (in radians, where 1 radian is equal to 57.295 779 513 1°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will exceed that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY299I QTNCT : ARG = argument, APPROACHES SINGULARITY

Explanation: In the subprogram IFYQTNCT (QTAN or QCOTAN), the argument value is too close to one of the singularities (±pi/2, ±3pi/2, for the tangent; ±pi, ±2pi, for the cotangent).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument (in radians where 1 radian is equivalent to 57.295 779 513 1°) to the trigonometric tangent or cotangent function is within the allowable range. If the argument may or will approach the corresponding singularities for the function during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY300I LGAMA : ARG = argument, (HEX = hexadecimal), LESS THAN OR EQUAL TO 2-252 OR GREATER THAN OR EQUAL TO 57.5744**

Explanation: In the subprogram IFYLGAMA (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.

IFY301I LGAMA : ARG = argument, (HEX = hexadecimal), LESS THAN OR EQUAL TO 0. OR GREATER THAN OR EQUAL TO 4.2937*1073**

Explanation: In the subprogram IFYLGAMA (DLGAMA), the value of the argument is outside the valid range (0 < x < 4.2937x10**73).

Supplemental Data Provided: The argument specified.

Standard Corrective Action: Result=•.

Programmer Response: Make sure that the argument to the DLGAMA function is within the allowable range. If the argument may or will be outside that range during program execution, then provide code to test for the situation and, if necessary, modify the argument or bypass the source referencing the function subprogram.

IFY900I 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.

IFY901I 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.

Refer to Chapter 10, "Extended Error Handling Subroutines and Error Option Table" on page 323 for information on the error handling subroutines.

IFY902I VEMG1 | VMOPT : ERROR NUMBER nnnn (REQUESTED BY MODULE module-name) IS OUT OF RANGE OF ERROR OPTION TABLE

Explanation: A request has been made to reference a nonexistent option table entry.

Supplemental Data Provided: The error number and module name.

System Action: The request is ignored, and execution continues. IRETCD is set to 0.

Refer to Chapter 10, "Extended Error Handling Subroutines and Error Option Table" on page 323 for information on the error handling subroutines.

Programmer Response: Make sure that the value assigned to an error condition is within the range of entries in the option table.

IFY903I VMOPT : 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.

Refer to Chapter 10, "Extended Error Handling Subroutines and Error Option Table" on page 323, for information on the error handling subroutines.

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 alter dynamically an unmodifiable entry in the option table.

IFY904I name : ATTEMPT TO DO I/O DURING FIXUP ROUTINE FOR AN I/O TYPE ERROR, FILE ffffff

Explanation: When attempting to correct an I/O error, the user may not issue another I/O statement or call a routine that issues an I/O statement.

Refer to Chapter 10, "Extended Error Handling Subroutines and Error Option Table" on page 323, for information on the error handling subroutines.

Supplemental Data Provided: The last 5 characters in the name of the module that issued the message: VSCOM, IBCOM, VIOFP, VIOCP, VIOUP, or VCOM2. The name of the file (fffffff).

System Action: The job step is terminated with a completion code of 16.

Programmer Response: Make sure that, if an I/O error is detected, the user exit routine does not attempt to execute any FORTRAN I/O statement.

IFY905I VSCOM | IBCOM | VCOM2 : SECONDARY ENTRY INTO MAIN ROUTINE, EXECUTION TERMINATED.

Explanation: A user program tried to enter a second time into a FORTRAN MAIN routine. Recursion is not allowed in VS FORTRAN.

Supplemental Data Provided: None.

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.

IFY906I VEMGN : ERROR NUMBER nnn, LINE NO. ll, REQUESTED BY MODULE mod-name HAS NO MESSAGE SKELETON.

Explanation: The text for line ll of error message number nnn could not be found in the message skeleton which is supposed to contain all such text.

Supplemental Data Provided: The error number (nnn), the line number (ll) of the message, and the name *(mod-name)* of the library module which 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.

Operator Messages

Operator messages for PAUSE and STOP statements may be generated during load module execution as follows:

yy IFY001A PAUSE x

Explanation: A VS FORTRAN 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.

IFY002I STOP x

Explanation: A VS FORTRAN 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.

IFY003I DSIOS ERROR OCCURRED ON FORTRAN OBJECT ERROR UNIT, PROGRAM TERMINATED.

Explanation: An error occurred on the VS FORTRAN object error unit.

Operator Response: None.

Appendix J. Library Module Names

Entry Name	Module Name
ABS	IFYFABS
ACOS	IFYSASCN
ADCON#	IFYVCVTH
AIMAG	IFYFIMAG
AINT	IFYFAINT
ALGAMA	IFYSGAMA
ALOG	IFYSLGN
ALOG10	IFYSLGC
AMAX0	IFYFMAXI
AMAX1	IFYFMAXR
AMIN0	IFYFMAXI
AMIN1	IFYFMAXR
AMOD	IFYFMODR
ANINT	IFYFNINT
ARCOS	IFYSASCN
ARSIN	IFYSASCN
ASIN	IFYSASCN
ATAN	IFYSATN2
ATAN2	IFYSATN2
BTEST	IFYBTSHS
CABS	IFYCSABS
CCMPR#	IFYCCMPR
CCOS	IFYCSSCN
CDABS	IFYCLABS
CDCOS	IFYCLSCN
CDDVD#	IFYCLAD
CDEXP	IFYCLEXP
CDLOG	IFYCLLOG
CDMPY#	IFYCLAM
CDSIN	IFYCLSCN
CDSQRT	IFYCLSQT
CDUMP	IFYVDUMP
CDVD#	IFYCSAD

Figure 58 (Part 1 of 10). Entry Names for Library Modules

Entry Name	Module Name
CERRST	IFYVCIAD
CEXP	IFYCSEXP
CHAR	IFYCITFN
CLOG	IFYCSLOG
CMOVE#	IFYCMOVE
CMPY#	IFYCSAM
CNCAT#	IFYCNCAT
CONJG	IFYFCONJ
COS	IFYSCOS
COSH	IFYSSCNH
COTAN	IFYSTNCT
CPDUMP	IFYVDUMP
CQABS	IFYCQABS
CQCOS	IFYCQSCN
CQDVD#	IFYCQRIT
CQEXP	IFYCQEXP
CQLOG	IFYCQLOG
CQMPY#	IFYCQRIT
CQSIN	IFYCQSCN
CQSQRT	IFYCQSQT
CSIN	IFYCSSCN
CSQRT	IFYCSSQT
CXMPR#	IFYCCMPR
DABS	IFYFABS
DACOS	IFYLASCN
DARCOS	IFYLASCN
DARSIN	IFYLASCN
DASIN	IFYLASCN
DATAN	IFYLATN2
DATAN2	IFYLATN2
DCONJG	IFYFCONJ
DCOS	IFYLCOS
DCOS	IFYWLCOS[1]
DCOSH	IFYLSCNH
DCOTAN	IFYLTNCT
DCOTAN	IFYWTNCT[1]
DDBDFLT	IFYDBDFT
DDIM	IFYFDIM
DEBUG#	IFYVDBUG
DEBUG#	IFYVDBUG
DERF	IFYLERF
DERFC	IFYLERF
DEXP	IFYLEXP
DEXP	IFYWLEXP[1]
DGAMMA	IFYLGAMA

Figure 58 (Part 2 of 10). Entry Names for Library Modules

Entry Name	Module Name
DIM	IFYFDIM
DIMAG	IFYFIMAG
DINT	IFYFAINT
DIOCS#	IFYDIOCS
DLGAMA	IFYLGAMA
DLOG	IFYLLGN
DLOG10	IFYLLGC
DMAX1	IFYFMAXD
DMIN1	IFYFMAXD
DMOD	IFYFMODR
DNINT	IFYFNINT
DPROD	IFYDPROD
DSIGN	IFYFSIGN
DSIN	IFYLSIN
DSIN	IFYWLSIN[1]
DSINH	IFYLSCNH
DSPAN#	IFYDSPAN
DSPN2#	IFYDSPAN
DSPN4#	IFYVSPAN
DSQRT	IFYLSQRT
DTAN	IFYLTNCT
DTAN	IFYWTNCT[1]
DTANH	IFYLTANH
DUMP	IFYVDUMP
DVCHK	IFYVDVCH
DYCMN#	IFYDDCMN
ERF	IFYSERF
ERFC	IFYSERF
ERRIAD	IFY3MOPT
ERRMON	IFYVMOPT
ERRMON	IFY3MOPT
ERRSAV	IFYVMOPT
ERRSAV	IFY3MOPT
ERRSET	IFYVMOPT
ERRSET	IFY3MOPT
ERRSTR	IFYVMOPT
ERRSTR	IFY3MOPT
ERRTRA	IFYVMOPT
ERRTRA	IFY3MOPT
EXIT	IFYVEXIT
EXP	IFYSEXP
EXP	IFYWSEXP[1]
FCDCD#	IFYFCDCD
FCDXI#	IFYFCDXI
FCQCQ#	IFYFCQCQ

Figure 58 (Part 3 of 10). Entry Names for Library Modules

Entry Name	Module Name
FCQXI#	IFYFCQXI
FCVAO	IFYVCVTH
FCVAO$	IFYVCVT$
FCVCO	IFYVCVTH
FCVCO$	IFYVCVT$
FCVDO	IFYVCVTH
FCVDO$	IFYVCVT$
FCVEO	IFYVCVTH
FCVEO$	IFYVCVT$
FCVIO	IFYVCVTH
FCVIO$	IFYVCVT$
FCVLO	IFYVCVTH
FCVLO$	IFYVCVT$
FCVQO	IFYVCVTH
FCVZO	IFYVCVTH
FCVZO$	IFYVCVT$
FCXPC#	IFYFCXPC
FCXPI#	IFYFCXPI
FDXPD#	IFYFDXPD
FDXPD#	IFYWDXPD[1]
FDXPI#	IFYFDXPI
FIXPI#	IFYFIXPI
FQTEN#	IFYVTEN
FQXPI#	IFYFQXPI
FQXPQ#	IFYFQXPQ
FQXP2#	IFYFQXPQ
FRDNL#	IFYNAMEL
FRDNL#	IFYNAMEL
FRXPI#	IFYFRXPI
FRXPR#	IFYFRXPR
FRXPR#	IFYWRXPR[1]
FTEN#	IFYVTEN
FWRNL#	IFYNAMEL
FWRNL#	IFYNAMEL
GAMMA	IFYSGAMA
IABS	IFYFABS
IAND	IFYBLOGL
IBCLR	IFYBTSHS
IBCOM#	IFYIBCOM
IBCOM#	IFYIBCOM
IBERH#	IFYVSERH
IBSET	IFYBTSHS
ICHAR	IFYCITFN
IDIM	IFYFDIM

Figure 58 (Part 4 of 10). Entry Names for Library Modules

Entry Name	Module Name
IDINT	IFYFIFIX
IDNINT	IFYFNINT
IEOR	IFYBLOGL
IFIX	IFYFIFIX
IFYCLCI0	IFYVLCI0
IFYCLCI0	IFYVLCI1
IFYCREN	IFYCREN
IFYCVIO$	IFYCVIO$
IFYDDCMP	IFYDDCMP
IFYDDCM0	IFYDDCM1
IFYDDIOS	IFYVDIOS
IFYDIOCS	IFYDIOCS
IFYDIOC0	IFYDIOC1
IFYDKIOS	IFYVKIOS
IFYDLBC1	IFYCLBC1
IFYDLCI0	IFYVLCI0
IFYDLCI0	IFYVLCI1
IFYDSPAP	IFYDSPAP
IFYDSPA0	IFYDSPA1
IFYFDXPD	IFYWDXPD
IFYFRXPR	IFYWRXPR
IFYIBCOP	IFYIBCOP
IFYIBCO0	IFYIBCO1
IFYLBCOM	IFYCLBC0
IFYLBCOM	IFYCLBC1
IFYLCOS	IFYWLCOS
IFYLDFIP	IFYLDFIP
IFYLDFI0	IFYLDFI1
IFYLSIN	IFYWLSIN
IFYLTNCT	IFYWTNCT
IFYNAMEP	IFYNAMEP
IFYNAME0	IFYNAME1
IFYQRFSW	IFYQERF
IFYSDUMQ	IFYSDUMQ
IFYSDUM0	IFYSDUM1
IFYSEXP	IFYWSEXP
IFYUATBL	IFYUATBL
IFYUOPT	IFYUOPT
IFYVASYP	IFYVASYP
IFYVASY0	IFYVASY1
IFYVBLN$	IFYVBLN$
IFYVBLNT	IFYVBLNT
IFYVCIA4	IFYVCIA4
IFYVCLMA	IFYVCLMI
IFYVCLOP	IFYVCLOP
IFYVCLO0	IFYVCLO1
IFYVCLSI	IFYVCLSI

Figure 58 (Part 5 of 10). Entry Names for Library Modules

Entry Name	Module Name
IFYVCNI$	IFYVCNI$
IFYVCNO$	IFYVCNO$
IFYVCOM$	IFYVCOM$
IFYVCOM2	IFYVCOM2
IFYVCONI	IFYVCONI
IFYVCONO	IFYVCONO
IFYVDBUP	IFYVDBUP
IFYVDBU0	IFYVDBU1
IFYVDIO$	IFYVDIO$
IFYVDUMQ	IFYVDUMQ
IFYVDUM0	IFYVDUM1
IFYVEMG1	IFYVEMGN
IFYVERE$	IFYVERE$
IFYVERM$	IFYVEMG$
IFYVERRE	IFYVERRE
IFYVERS$	IFYVERS$
IFYVFNTH	IFYVFNTH
IFYVGMFM	IFYVGMFM
IFYVIIO$	IFYVIIO$
IFYVIIOS	IFYVIIOS
IFYVINQP	IFYVINQP
IFYVINQ0	IFYVINQ1
IFYVIOCP	IFYVIOCP
IFYVIOC0	IFYVIOC1
IFYVIOD0	IFYVIOD1
IFYVIOD1	IFYVIOD0
IFYVIOFP	IFYVIOFP
IFYVIOF0	IFYVIOF1
IFYVIOI0	IFYVIOI1
IFYVIOI1	IFYVIOI0
IFYVIOK0	IFYVIOK1
IFYVIOK1	IFYVIOK0
IFYVIOLP	IFYVIOLP
IFYVIOL0	IFYVIOL1
IFYVIONP	IFYVIONP
IFYVION0	IFYVION1
IFYVIOUP	IFYVIOUP
IFYVIOU0	IFYVIOU1
IFYVKIO$	IFYVKIO$
IFYVLBC1	IFYCLBC1
IFYVLCIN	IFYVLCI0
IFYVLCIN	IFYVLCI1
IFYVLCI0	IFYVLCI1
IFYVLCI1	IFYVLCI0
IFYVLINP	IFYVLINP
IFYVLIN0	IFYVLIN1

Figure 58 (Part 6 of 10). Entry Names for Library Modules

Entry Name	Module Name
IFYVLOAD	IFYVLOAD
IFYVLOC$	IFYVLOC$
IFYVLOCA	IFYVLOCA
IFYVMOPT	IFYVMOPT
IFYVMOP0	IFYVMOP1
IFYVMOP4	IFYVMOPP
IFYVMSKL	IFYVMSKL
IFYVOPEP	IFYVOPEP
IFYVOPE0	IFYVOPE1
IFYVPARM	IFYVPARM
IFYVPOS$	IFYVPOS$
IFYVPOSA	IFYVPOSA
IFYVPOST	IFYVPOST
IFYVSCOP	IFYVSCOP
IFYVSCO0	IFYVSCO1
IFYVSFIO	IFYVSFIO
IFYVSFST	IFYVSFST
IFYVSIOS	IFYVSIOS
IFYVSPAP	IFYVSPAP
IFYVSPA0	IFYVSPA1
IFYVSPIE	IFYVSPIE
IFYVSTA$	IFYVSTA$
IFYVSTAE	IFYVSTAE
IFYVTRC$	IFYVTRC$
IFYVTRCH	IFYVTRCH
IFYVVDIR	IFYCVIOS
IFYVVSEQ	IFYCVIOS
IN#	IFYVASYN
INDEX	IFYINDEX
INT	IFYFIFIX
IOR	IFYBLOGL
ISHFT	IFYBTSHS
ISIGN	IFYFSIGN
IXCCMSD	IFYVCMSS
LDFIO#	IFYLDFIO
LEN	IFYCITFN
LGAMMA	IFYSGAMA
LGE	IFYLXCMP
LGT	IFYLXCMP
LLE	IFYLXCMP
LLT	IFYLXCMP
LOG	IFYSLGN
LOG10	IFYSLGC
MAX0	IFYFMAXI
MAX1	IFYFMAXR

Figure 58 (Part 7 of 10). Entry Names for Library Modules

Entry Name	Module Name
MIN0	IFYFMAXI
MIN1	IFYFMAXR
MOD	IFYFMODI
NINT	IFYFNINT
NOT	IFYBLOGL
OUT#	IFYVASYN
OVERFL	IFYVOVER
PDUMP	IFYVDUMP
QABS	IFYFABS
QARCOS	IFYQASCN
QARSIN	IFYQASCN
QATAN	IFYQATN2
QATAN2	IFYQATN2
QCONJG	IFYFCONJ
QCOS	IFYQSCN
QCOSH	IFYQSCNH
QCOTAN	IFYQTNCT
QDIM	IFYFDIM
QDTAN	IFYLTNCT
QDTAN	IFYWTNCT
QERF	IFYQERF
QERFC	IFYQERF
QERF2	IFYQERF2
QEXP	IFYFQXPQ
QIMAG	IFYFIMAG
QINT	IFYFAINT
QLOG	IFYFQXPQ
QLOG10	IFYFQXPQ
QMOD	IFYFMODR
QSIGN	IFYFSIGN
QSIN	IFYQSCN
QSINH	IFYQSCNH
QSQRT	IFYQSQRT
QTAN	IFYQTNCT
QTANH	IFYQTANH
SDUMP	IFYSDUMP
SIGN	IFYFSIGN
SIN	IFYSSIN
SINH	IFYSSCNH
SQRT	IFYSSQRT

Figure 58 (Part 8 of 10). Entry Names for Library Modules

Entry Name	Module Name
TAN	IFYSTNCT
TANH	IFYSTANH
TFORT#	IFYTFORT
VCEND#	IFYVCIAD
VCINT#	IFYVCIAD
VCIORT	IFYVCIAD
VCLSE#	IFYVCLOS
VCOMHALT	IFYVCOMH
VDCON#	IFYVCIAD
VFCB#	IFYVIOCT
VFCD#	IFYVIOCT
VFCE#	IFYVIOCT
VFCR#	IFYVIOCT
VFCSF#	IFYVIOFM
VFCSL#	IFYVIOLD
VFCSN#	IFYVIONL
VFCSN#	IFYVIONL
VFDSF#	IFYVIOFM
VFDSL#	IFYVIOLD
VFEE#	IFYVINTE
VFEIM#	IFYVINTE
VFEIN#	IFYVINTE
VFELC#	IFYVINTE
VFEP#	IFYVINTE
VFES#	IFYVINTE
VFESF#	IFYVIOFM
VFESL#	IFYVIOLD
VFESN#	IFYVIONL
VFFDU#	IFYVIOUF
VFFXF#	IFYVIOFM
VFFXL#	IFYVIOLD
VFFXU#	IFYVIOUF
VFIXF#	IFYVIOFM
VFIXL#	IFYVIOLD
VFIXU#	IFYVIOUF
VFLIM#	IFYVLINK
VFLIS#	IFYVLINK
VFQKF#	IFYVIOFM
VFQKU#	IFYVIOUF
VFRDF#	IFYVIOFM
VFRDU#	IFYVIOUF
VFRIF#	IFYVIOFM
VFRIL#	IFYVIOLD
VFRIN#	IFYVIONL
VFRKF#	IFYVIOFM
VFRKU#	IFYVIOUF

Figure 58 (Part 9 of 10). Entry Names for Library Modules

Entry Name	Module Name
VFRSF#	IFYVIOFM
VFRSL#	IFYVIOLD
VFRSN#	IFYVIONL
VFRSU#	IFYVIOUF
VFSXF#	IFYVIOFM
VFSXU#	IFYVIOUF
VFUVF#	IFYVIOFM
VFUVU#	IFYVIOUF
VFWDF#	IFYVIOFM
VFWDU#	IFYVIOUF
VFWIF#	IFYVIOFM
VFWIL#	IFYVIOLD
VFWIN#	IFYVIONL
VFWKF#	IFYVIOFM
VFWKU#	IFYVIOUF
VFWSF#	IFYVIOFM
VFWSL#	IFYVIOLD
VFWSN#	IFYVIONL
VFWSU#	IFYVIOUF
VINQR#	IFYVINQR
VLDIO#	IFYLDFIO
VOPEN#	IFYVOPEN
VRDNL#	IFYNAMEL
VSCOM#	IFYVSCOM
VSERH#	IFYVSERH
VWRNL#	IFYNAMEL
WAIT#	IFYVASYN
XUFLOW	IFYVXMSK

Figure 58 (Part 10 of 10). Entry Names for Library Modules

Note to 58

1 Alternative mathematical library subroutine module name

OS	CMS2	DOS
IFYVRENT	IFYVRENT	IFYVRENT
IFYVCOMH	IFYVCOMH	IFYVCOMH
IFYVSIOS	IFYVSIOS	IFYDSIOS
IFYVDIOS	IFYVDIOS	IFYDDIOS
IFYVIIOS	IFYVIIOS	IFYVIIOS
IFYVVIOS	IFYCVIOS	IFYDVIOS
IFYVCVTH	IFYVCVTH	IFYVCVTH
IFYVCONI	IFYVCONI	IFYVCONI
IFYVCONO	IFYVCONO	IFYVCONO
IFYVTEN	IFYVTEN	IFYVTEN
IFYVERRM	IFYVERRM	IFYVERRM
IFYVERRE	IFYVERRE	IFYVERRE
IFYVTRCH	IFYVTRCH	IFYVTRCH

Figure 59. Reentrant Library Module Names

Glossary

This glossary includes definitions developed by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO).

An asterisk (*) to the left of a term indicates that the entire entry is reproduced from the *American National Dictionary for Information Processing*, copyright 1977 by the Computer and Business Equipment Manufacturers Association, copies of which may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

An asterisk (*) to the right of an item number indicates an ANSI definition in an entry that also includes other definitions.

The symbol "(ISO)" at the beginning of a definition indicates that it has been discussed and agreed upon at meetings of the International Organization for Standardization Technical Committee 97/Subcommittee 1 (Data Processing Vocabulary), and has also been approved by ANSI and included in the *American National Dictionary for Information Processing*.

•. • is used in this manual to represent the maximum floating-point value.

alphabetic character. A character of the set A, B, C,...,Z. See also "letter."

IBM Extension

In VS FORTRAN, the currency symbol ($) is considered an alphabetic character.

End of IBM Extension

alphameric. Pertaining to a character set that contains letters (A through Z) and digits (0 through 9) only.

alphameric character set. A character set that contains both letters and digits.

argument. A parameter passed between a calling program and a SUBROUTINE subprogram, a FUNCTION subprogram, or a statement function.

arithmetic constant. A constant of type integer, real, double precision, or complex.

arithmetic expression. One or more arithmetic operators and/or arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, array element, or function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator. A symbol that directs VS FORTRAN to perform an arithmetic operation. The arithmetic operators are:

+ addition
- subtraction
* multiplication
/ division
** exponentiation.

array. An ordered set of data items identified by a single name.

array declarator. The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension. An array declarator may appear in a DIMENSION, COMMON, or explicit type statement.

array element. A data item in an array, identified by the array name followed by a subscript indicating its position in the array.

array name. The name of an ordered set of data items that make up an array.

assignment statement. A statement that assigns a value to a variable or array element. It is made up of a variable or array element, followed by an equal sign (=), followed by an expression. The variable, array element, or expression can be character, logical, or arithmetic. When the assignment statement is executed, the expression to the right of the equal sign replaces the value of the variable or array element to the left.

basic real constant. A string of decimal digits containing a decimal point, and expressing a real value.

blank common. An unnamed common block.

character constant. A string of one or more alphameric characters enclosed in apostrophes. The delimiting apostrophes are not part of the constant.

character expression. An expression in the form of a single character constant, variable, array element, substring, function reference, or another expression enclosed in parentheses. A character expression is always of type character.

character type. A data type that can consist of any alphameric characters; in storage, one byte is used for each character.

common block. A storage area that may be referred to by a calling program and one or more subprograms.

complex constant. An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first real constant of the pair is the real part of the complex number; the second is the imaginary part.

complex type. An approximation of the value of a complex number, consisting of an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

connected file. A file that has been connected to a unit and defined by a FILEDEF command or by job control statements.

constant. An unvarying quantity. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and hexadecimal data.

control statement. Any of the statements used to alter the normal sequential execution of VS FORTRAN statements, or to terminate the execution of a VS FORTRAN program. FORTRAN control statements are any of the forms of the GO TO, IF, and DO statements, or the PAUSE, CONTINUE, and STOP statements.

data. (1)* (ISO) A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. (2) In FORTRAN, data includes constants, variables, arrays, and character substrings.

data item. A constant, variable, array element, or character substring.

data set. The major unit of data storage and retrieval consisting of data collected in one of several prescribed arrangements and described by control information to which the system has access.

data set reference number. A constant or variable in an input or output statement that identifies a data set to be processed; the unit number.

data type. The properties and internal representation that characterize data and functions. The basic types are integer, real, complex, logical, double precision, and character.

*** digit.** (ISO) A graphic character that represents an integer. For example, one of the characters 0 to 9.

DO loop. A range of statements executed repetitively by a DO statement. See also "range of a DO."

DO variable. A variable, specified in a DO statement, that is initialized or incremented prior to each execution of the statement or statements within a DO range. It is used to control the number of times the statements within the range are executed. See also "range of a DO."

double precision. The standard name for real data of storage length 8.

dummy argument. A variable within a subprogram or statement function definition with which actual arguments from the calling program or function reference are positionally associated. Dummy arguments are defined in a SUBROUTINE or FUNCTION statement, or in a statement function definition.

executable program. A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures, or both.

executable statement. A statement that causes an action to be taken by the program; for example, to calculate, to test conditions, or to alter normal sequential execution.

existing file. A file that has been defined by a FILEDEF command or by job control statements. A valid unit number in FORTRAN's internal unit assignment table, as specified at installation time. (RPC 11992)

The INQUIRE statement considers a file to exist on the basis of VS FORTRAN I/O statements that have been processed (RPC17664)

existing unit. A valid unit number in FORTRAN's internal unit assignment table, as specified at installation. (RPC11970)

expression. A notation that represents a value: a constant or a reference appearing alone, or combinations of constants and/or references with operators. An expression can be arithmetic, character, logical, or relational.

external file. A set of related external records treated as a unit; for example, in stock control, an external file would consist of a set of invoices.

external function. A function defined outside the program unit that refers to it.

external procedure. A SUBROUTINE or FUNCTION subprogram written in FORTRAN.

file. A set of records. If the file is located in internal storage, it is an internal file; if it is on an input/output device, it is an external file.

file definition statement. A statement that describes the characteristics of a file to a user program. For example, the OS/VS DD statement or DOS/VSE ASSGN statement for batch processing, or the FILEDEF command for CMS processing.

file reference. A reference within a program to a file. It is specified by a unit identifier.

formatted record. (1) A record, described in a FORMAT statement, that is transmitted, when necessary with data conversion, between internal storage and internal storage or to an external record. (2) A record transmitted with list-directed READ or WRITE statements and an EXTERNAL statement.

FORTRAN-supplied procedure. See "intrinsic function."

function reference. A source program reference to an intrinsic function, to an external function, or to a statement function.

function subprogram. A subprogram invoked through a function reference, and headed by a FUNCTION statement. It returns a value to the calling program unit at the point of reference.

IBM Extension

hexadecimal constant. A constant that is made up of the character Z followed by two or more hexadecimal digits.

End of IBM Extension

hierarchy of operations. The relative order used to evaluate expressions containing arithmetic, logical, or character operations.

implied DO. An indexing specification (similar to a DO statement, but without specifying the word DO) with a list of data elements, rather than a set of statements, as its range. In a DATA statement the list can contain integer constants or expressions containing integer constants. In input/output statements the list can contain integer, real, or double precision arithmetic expressions.

integer constant. A string of decimal digits containing no decimal point and expressing a whole number.

integer expression. An arithmetic expression whose values are of integer type.

integer type. An arithmetic data type capable of expressing the value of an integer. It can have a positive, negative, or zero value. It must not include a decimal point.

internal file. A set of related internal records treated as a unit.

intrinsic function. A function, supplied by VS FORTRAN, that performs mathematical or character operations.

* **I/O.** Pertaining to either input or output, or both.

I/O list. A list of variables in an input or output statement specifying which data is to be read or which data is to be written. An output list may also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

labeled common. See "named common."

length specification. A source language specification of the number of bytes to be occupied by a variable or an array element.

letter. A symbol representing a unit of the alphabet.

list-directed. An input/output specification that uses a data list instead of a FORMAT specification.

logical constant. A constant that can have one of two values: true or false.

logical expression. A combination of logical primaries and logical operators. A logical expression can have one of two values: true or false.

logical operator. Any of the set of operators .NOT. (negation), .AND. (connection: both), or .OR. (inclusion: either or both), .EQV. (equal), .NEQV. (not equal).

logical primary. A primary that can have the value true or false. See also "primary."

logical type. A data type that can have the value true or false for VS FORTRAN. See also "data type."

looping. Repetitive execution of the same statement or statements. Usually controlled by a DO statement.

main program. A program unit, required for execution, that can call other program units but cannot be called by them.

name. A string of from one through six alphameric characters, the first of which must be alphabetic. Used to identify a constant, a variable, an array, a function, a subroutine, or a common block.

named common. A separate common block consisting of variables, arrays, and array declarators, and given a name.

nested DO. A DO statement whose range is entirely contained within the range of another DO statement.

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.

nonexisting file. A file that has not been defined by a FILEDEF command or by job control statements.

*** numeric character.** (ISO) Synonym for digit.

numeric constant. A constant that expresses an integer, real, or complex number.

preconnected file. A unit or file that was defined at installation time. However, a preconnected file does not exist for a program if the file is not defined by a FILEDEF command or by job control statements.

predefined specification. The implied type and length specification of a data item, based on the initial character of its name in the absence of any specification to the contrary. The initial characters I through N type data items as integer; the initial characters A through H, O through Z, and $ type data items as real. No other data types are predefined. For VS FORTRAN, the length for both types is 4 bytes.

primary. An irreducible unit of data; a single constant, variable, array element, function reference, or expression enclosed in parentheses.

procedure. A sequenced set of statements that may be used at one or more points in one or more computer programs, and that usually is given one or more input parameters and returns one or more output parameters. A procedure consists of subroutines, function subprograms, and intrinsic functions.

procedure subprogram. A function or subroutine subprogram.

program unit. A sequence of statements constituting a main program or subprogram.

range of a DO. Those statements that physically follow a DO statement, up to and including the statement specified by the DO statement as being the last to be executed; also called a "DO loop."

real constant. A string of decimal digits that expresses a real number. A real constant must contain either a decimal point or a decimal exponent and may contain both.

real type. An arithmetic data type, capable of approximating the value of a real number. It can have a positive, negative, or zero value.

record. A collection of related items of data treated as a unit.

relational expression. An expression that consists of an arithmetic expression, followed by a relational operator, followed by another arithmetic expression or a character expression, followed by a relational operator, followed by another character expression. The result is a value that is true or false.

relational operator. Any of the set of operators:

.GT. greater than
.GE. greater than or equal to
.LT. less than
.LE. less than or equal to
.EQ. equal to
.NE. not equal to

scale factor. A specification in a VS FORTRAN FORMAT statement that changes the location of the decimal point in a real number (and, on input, if there is no exponent, the magnitude of the number).

specification statement. One of the set of statements that provides the compiler with information about the data used in the source program. In addition, the statement supplies the information required to allocate data storage.

specification subprogram. A subprogram headed by a BLOCK DATA statement and used to initialize variables in named common blocks.

statement. The basic unit of a VS FORTRAN program, that specifies an action to be performed, or the nature and characteristics of the data to be processed, or information about the program itself. Statements fall into two broad classes: executable and nonexecutable.

statement function. A name, followed by a list of dummy arguments, that is equated to an arithmetic, logical, or character expression. In the remainder of the program the name can be used as a substitute for the expression.

statement function definition. A statement that defines a statement function. Its form is a name, followed by a list of dummy arguments, followed by an equal sign (=), followed by an arithmetic, logical, or character expression.

statement function reference. A reference in an arithmetic, logical, or character expression to the name of a previously defined statement function.

statement label. A number of from one through five decimal digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a DO, or to refer to a FORMAT statement.

statement number. See "statement label."

subprogram. A program unit that is invoked by another program unit in the same program. In VS FORTRAN, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

subroutine subprogram. A subprogram whose first statement is a SUBROUTINE statement. It optionally returns one or more parameters to the calling program unit.

*** subscript.** (1) (ISO) A symbol that is associated with the name of a set to identify a particular subset or element.

(2) A subscript quantity or set of subscript quantities, enclosed in parentheses and used with an array name to identify a particular array element.

subscript quantity. A component of a subscript: an integer constant, an integer variable, or an expression evaluated as an integer constant.

IBM Extension

In VS FORTRAN, a subscript quantity may also be a real constant, variable, or expression.

End of IBM Extension

type declaration. The explicit specification of the type of a constant, variable, array, or function by use of an explicit type specification statement.

unformatted record. A record that is transmitted unchanged between internal storage and an external record.

unit. A means of referring to a file in order to use input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

unit identifier. The number that specifies an external unit.

1. An integer expression whose value must be zero or positive. For VS FORTRAN, this integer value of length 4 must correspond to a DD name, a FILEDEF name, or an ASSGN name.

2. An asterisk (*) that corresponds on input to FT05F001 and on output to FT06F001.

3. The name of a character array, character array element, or character substring for an internal file.

variable. (1) * A quantity that can assume any of a given set of values.

(2) A data item, identified by a name, that is not a named constant, array, or array element, and that can assume different values at different times during program execution.

Index

Special Characters

. (period) 11
... (ellipsis) ix
+ (plus sign) 11
$ (currency symbol)
  source statement characters 11
* (asterisk)
  READ statement 192
  source statement characters 11
  WRITE statement 274
- (minus sign or hyphen) 11
- (minus sign) 9
/ (slash) 11
, (comma) 11
( ) (parentheses) 11
: (colon) 11
[ ] (brackets) viii
' (apostrophe) 11
= (equal sign) 11
" (quotation mark) 9, 11
•
  See Glossary for definition 545

A

A format code 125
ABS/IABS
  error message 513
  storage estimate 417
ABS/IABS/DABS/QABS
  algorithm 372
  description 292
  registers used 439
absolute value subprograms
  algorithms 372
  description 292
  registers used 439
ACCESS=
  INQUIRE by file name 157
  INQUIRE by unit number 162
  OPEN statement 170
accuracy statistics 425-432
ACOS
  See ASIN/ACOS
ACTION=
  INQUIRE by file name 158
  INQUIRE by unit number 162
  OPEN statement 170
actual argument 30
  array name 30
  in a function subprogram 140
  in a subroutine subprogram 239
  in an ENTRY statement 100
AIMAG
  description 294
  registers used 439
  storage estimate 417
AINT
  description 294
  registers used 439
  storage estimate 417
ALGAMA
  See GAMMA/ALGAMA 293
algorithms 369
ALOG/ALOG10
  accuracy 427
  algorithm 394
  description 288
  effect of an argument error 395
  error message 513
  registers used 439
  storage estimate 417, 418
alphabetic character
  See also letter
  definition 545
alphabetic character set 11
alphabetic primary
  See primary
alphameric character set
  definition 545
alphameric, definition 545
alternate return specifier 100
alternative mathematical library subroutines
  explicitly called mathematical subprograms 438
  module names 533
  storage estimate 417
alternative paths of execution 145
AMOD/DMOD
  description 294
  registers used 439
  storage estimate 417
ANINT
  description 295
  registers used 439
  storage estimate 417
ANS FORTRAN features 351
ANSI definitions 545
apostrophe 11
arccosine subprograms
  algorithms 373
  size 438
arcsine subprograms
  algorithms 373
  size 438
arctangent subprograms
  algorithms 376
  size 438
argument
  actual 100, 239

definition 545
dummy 100, 239
arguments
assembler language 435
implicitly called 314-319
range of accuracy 425
arithmetic assignment statement 61
conversion rules (complex) 64
conversion rules (integer or real) 63
description of 61
valid statements 64
arithmetic constant
See also digit
complex 19
definition 545
integer 16
primary 36
real 17
arithmetic expression 36
definition 545
description of 36
rules for constructing 37
type and length of (complex) 42
type and length of (integer) 40
type and length of (real) 41
use of parentheses in 39
arithmetic IF statement 145
arithmetic operation 37
addition 36, 37
division 36, 37
evaluation of functions 37
exponentiation 36, 37
first operand, complex 39
first operand, integer 39
first operand, real 39
multiplication 36, 37
rules for constructing 37
subtraction 36, 37
unary 36
arithmetic operator 36
definition 545
description of 36
operations involving 49
arithmetic subprograms, modular
size 438
array
actual argument 30
assumed-size 31
declarator 28
declarator, definition 545
definition 545
description of 187
dimension bounds 30
DIMENSION statement 87
dimensions of 87
dummy argument 31
element, definition 545
element, invalid 29
element, valid 29
elements 28
name
READ statement 183, 187
WRITE statement 251, 255
name, definition 545
READ statement 183
size and type declaration 30, 31
specifications 187
subscripts 28
array dimension error message 492
ASCII/ISCII codes 361
ASIN/ACOS
accuracy 427
algorithm 373
description 290
effect of an argument error 373
error message 514
registers used 439
storage estimate 417
assembler language
calling sequence 433
requirements 442
assign a name to a constant 173
assign a name to a main program 177
assign a number to a variable 59
ASSIGN statement 59
assigned GO TO statement 143
assignment statement 60
arithmetic 61
ASSIGN statement 59
character 61
definition 545
description of 59
logical 61
associate actual with dummy argument 70
assumed length 105
assumed-size array 31
asterisk
READ statement 192
source statement characters 11
WRITE statement 274
asterisks in formatted output
D, E, and Q format codes 118
example 124
I format code 116
meaning of 113
asynchronous
READ statement 179
WRITE statement 247
asynchronous I/O error message 485
asynchronous I/O subtask 497
AT statement 66
description of 66
in debug packet 83
ATAN/ATAN2
accuracy 427
algorithm 376
description 290
effect of an argument error 376
error message 514
registers used 439

storage estimate 417

B

BACKSPACE statement 67
description of 87
invalid statements 68
valid statements 67
basic real constant
definition 545
description of 17
begin debug packet 66
bit function error message 485
bit manipulation functions 300
bit manipulation subprograms
assembler information 441
manipulation routines 313
storage estimates 424
blank 11
common blocks 78
common blocks, definition 78
common, definition 545
format code 131
FORMAT statement 130
INQUIRE by file name 157
INQUIRE by unit number 162
named common blocks 78
source statement characters 11
BLANK=
INQUIRE by file name 157
INQUIRE by unit number 162
OPEN statement 170
BLOCK DATA statement 69
block data subprogram 56
block IF statement 146
ELSE 148
ELSE IF 148
END IF 147
BN format code 130
BTEST
description 300
storage estimate 422
BTSHS error message 485
bypass statements 79
BZ format code 131

C

CABS/CDABS
accuracy 427
algorithm 372
description 292
registers used 439
storage estimate 417
CALL CDUMP 317
CALL DUMP 315
CALL macro instruction 434
CALL SDUMP 320
CALL statement 70
CDUMP/CPDUMP 317
description of 70
DUMP/PDUMP 315
DVCHK 313
ERRMON 324
ERRSAV 325
ERRSET 325
ERRSTR 329
ERRTRA 329
EXIT 317
OPSYS 318
OVERFL 313
SDUMP 319
XUFLOW 322
calling sequence
in assembler language 435, 441
calling VS FORTRAN subprograms
explicitly 372
in assembler language 434
carrier control
H format code 127
T format code 128
CCMPR#
description 310
error message 493
storage estimate 421
CCOS
See CSIN/CCOS
CDABS
See CABS/CDABS
CDCOS
See CDSIN/CDCOS
CDDVD#/CDMPY#
algorithm 410
description 309
effect of an argument error 410
registers used 440
storage estimate 418
CDEXP
See CEXP/CDEXP
CDLOG
See CLOG/CDLOG
CDMPY#
See CDDVD#/CDMPY#
CDSIN/CDCOS
accuracy 427
algorithm 399
description 290
effect of an argument error 400
error message 523
registers used 439
storage estimate 417
CDSQRT
See CSQRT/CDSQRT
CDUMP/CPDUMP
See CDVD#/CMPY#
CDVD#/CMPY#
algorithm 410

description 309
effect of an argument error 410
registers used 440
storage estimate 418
CEXP/CDEXP
accuracy 427
algorithm 386
description 288
effect of an argument error 386
error message 519, 523
registers used 439
storage estimate 418
change options 325
CHAR
description 299
storage estimate 421
character array element
READ statement 183, 187
WRITE statement 251, 255
character array name
READ statement 183, 187
WRITE statement 251, 255
character assignment statement 61
character constant 21
definition 546
description of 21
READ statement 183, 187
transmission 127
valid 22
WRITE statement 251, 255
character constant data
example 124
FORMAT statement 112
character data
dump 317
transmission 125
character expression 44
definition 546
description of 44
READ statement 183, 187
use of parentheses in 44
WRITE statement 251, 255
character manipulation functions 299
character manipulation subprograms
assembler information 440
error messages 492, 498
storage estimate 421
character operator
character expressions 44
operations involving 49
character skipping 128
character string, inherited length of 105
character substring 32
description of 32
reference 32
variable 32
character type
definition 546
IMPLICIT type statement 151
in function subprogram 99
CHARACTER type statement 103
character variable
storage length 25
substring 32
CITFN error message 493
CLOG/CDLOG
accuracy 427
algorithm 396
description 288
effect of an argument error 396
error message 519, 523
registers used 439
storage estimate 418
CLOSE statement 74
description of 74
examples 75
CLOSE statement error message 488, 492
CMOVE#
description 310
error message 494
storage estimate 421
CMPY#
See CDVD#/CMPY#
CNCAT#
description 310
error message 495
storage estimate 421
colon
format code 132
source statement characters 11
comma 11
comments
fixed-form 7, 76
free-form 9, 76
common block
definition 546
to initialize variables 69
common logarithmic subprograms
algorithms 394
size 438
COMMON statement 77
compare complex numbers 309
compiler-directed statement 57
description of 57
EJECT 94
INCLUDE 153
compiler, executing on 3
complex constant 19
definition 19, 546
invalid 20
valid 20
complex data requirements 111
complex exponentiation subprograms 426, 432
COMPLEX format 315, 448
complex multiply and divide subprograms 416
complex type 151
definition 546
explicit type statement 103
IMPLICIT type statement 151
COMPLEX type statement 103

complex variable
  storage length 25
complex-to-integer 309
compop 310
computed GO TO statement 144
concatenation operand error message 495
COND=
  WAIT statement 244
CONJG
  description 294
  registers used 439
  storage estimate 418
conjugate of a complex number subprograms
  size 438
connect file to unit 168
connected file
  definition 546
  formatted READ—direct access 184
  formatted READ—sequential access 194
  formatted WRITE—sequential access 260
  INQUIRE by file name 156
  READ with list-directed I/O 211
  READ with NAMELIST 217
  unformatted READ—direct access 197
  unformatted READ—sequential access 204
  unformatted WRITE—direct access 263
  WRITE with list-directed I/O 275
  WRITE with NAMELIST 281
constant 15
  arithmetic 16
  assign a name to 173
  character 21
  complex 19
  definition 546
  hexadecimal 23
  Hollerith 22
  integer 16
  iref refid=zfoc020.classes 15
  logical 21
  real 17
continuation line
  fixed-form 8
  free-form 10
continue a DO loop 79
CONTINUE statement 79
continued line 9
  free-form 9
control program exceptions 371
control statement 54
  assigned GO TO 143
  CALL 70
  computed GO TO 144
  CONTINUE 79
  definition 546
  description of 54
  DO 89
  END statement 94
  GO TO 142
  IF 145
  PAUSE 175
  RETURN 222
  STOP 237
  unconditional GO TO 144
conversion functions 296
conversion rules 63
corrective action
  after error 334
  after mathematical subroutine error 341
  after program interrupt 341
COS
  See SIN/COS
COSH
  See SINH/COSH
cosine subprograms
  algorithms 398
COTAN
  See TAN/COTAN
cotangent subprograms
  algorithms 411
CQABS
  accuracy 428
  algorithm 372
  description 292
  effect of an argument error 372
  registers used 439
  storage estimate 418
CQCOS
  See CQSIN/CQCOS
CQDVD#
  See CQMPY#/CQDVD#
CQEXP
  accuracy 428
  algorithm 387
  description 288
  effect of an argument error 387
  error message 521
  registers used 439
  storage estimate 418, 419
CQLOG
  accuracy 428
  algorithm 397
  description 288
  effect of an argument error 397
  registers used 439
  storage estimate 418, 419
CQMPY#/CQDVD#
  accuracy 428
  algorithm 410
  description 309
  effect of an argument error 410
  registers used 440
  storage estimate 418, 419
CQSIN/CQCOS
  accuracy 428
  algorithm 401
  description 290
  effect of an argument error 402
  error message 522
  registers used 439
  storage estimate 418
CQSQRT
  accuracy 428

algorithm 406
description 293
effect of an argument error 406
registers used 439
storage estimate 418
create a file 168
create a preconnected file 168
CSIN/CCOS
accuracy 427, 428
algorithm 399
description 290
effect of an argument error 399
error message 520
registers used 439
storage estimate 417
CSQRT/CDSQRT
accuracy 427, 428
algorithm 403, 405
description 293
effect of an argument error 404
registers used 439
storage estimate 418
currency symbol 11
IMPLICIT statement 152
CVIOS error messages 485, 500, 504
CXMPR#
description 309
registers used 440
storage estimate 421

D

D format code 117
DABS
See ABS/IABS/DABS/QABS
storage estimate 417
DACOS
See DASIN/DACOS
DASIN/DACOS
accuracy 428
algorithm 374
description 290
effect of an argument error 374
error message 517
registers used 439
storage estimate 418
data
definition 15, 546
item, definition 546
transfer 128
type, definition 546
data set
reference number, definition 546
DATA statement 54
character data in 81
description of 54, 80
implied DO in 91
DATAN/DATAN2
accuracy 428
algorithm 377
description 290
effect of an argument error 377
error message 517
registers used 439
storage estimate 418
DCONJG
description 294
registers used 439
storage estimate 418
DCOS
See DSIN/DCOS
DCOSH
See DSINH/DCOSH
DCOTAN
See DTAN/DCOTAN
DDIM
description 295
registers used 439
DDIOS error message 501
debug 82
DEBUG statement 54, 82
AT statement 66, 83
description of 54, 82
DISPLAY statement 83, 88
END DEBUG statement 83, 95
examples 84
SUBCHK function 84
TRACE OFF statement 83, 243
TRACE ON statement 83, 243
decimal point in format codes 113
declaration of type 27
default options 334
define values of
array elements 80, 103
arrays 80, 103
substrings 80
variables 80, 103
definitions
DELETE statement 87
deleting records 87
DERF/DERFC
accuracy 428
algorithm 381
description 292
effect of an argument error 382
registers used 439
storage estimate 418
description of
DEXP
accuracy 429
algorithm 386
description 288
effect of an argument error 386
error message 516
registers used 439
storage estimate 418, 419
DFNTH error message 487
DGAMMA/DLGAMA

accuracy 429
algorithm 389
description 293
effect of an argument error 390
error message 529
registers used 439
storage estimate 418
digit
definition 546
source statement characters 11
DIM
description 295
registers used 439
storage estimate 418
DIMAG
description 294
registers used 439
storage estimate 417
dimension bound
lower 30
description 30
DIMENSION statement 88
explicit statement 105
upper 30
description 30
DIMENSION statement 88
explicit statement 105
DIMENSION statement 87
DINT
description 294
registers used 439
storage estimate 417
DIOCS error message 505
direct access
files 170
input/output 162
INQUIRE statement 156, 161
READ statement 182, 196
WRITE statement 250, 262
direct access file error message 468
DIRECT=
INQUIRE by file name 156
INQUIRE by unit number 161
disconnect an external file 74
display data in NAMELIST format 88
DISPLAY statement 88
description of 88
in debug packet 83
divide complex numbers 309
divide-check exception test 313
divide-check service subprogram 369
See also DVCHK
error message 466
DLGAMA
See DGAMMA/DLGAMA
DLOG/DLOG10
accuracy 429
algorithm 395
description 288
effect of an argument error 395
error message 516
registers used 439
storage estimate 418, 419
DMOD
See AMOD/DMOD
DNINT
description 295
registers used 439
storage estimate 417
DO list 80
DO loop 83, 89
See also range of a DO
definition 546
DO statement 89
DO variable
definition 546
implied in DATA statement 91
implied in input/output statement 92
double precision 26
constant 20
data editing 117
definition 546
in statements 26
storage length 25
type 103, 151
DOUBLE PRECISION type statement 103
DP
assign 64
extend 64
float 64
DPROD
description 295
registers used 439
storage estimate 419
DSIGN
description 295
registers used 439
storage estimate 419
DSIN/DCOS
accuracy 428, 429
algorithm 399
description 290
effect of an argument error 399
error message 516
registers used 439
storage estimate 417, 418
DSINH/DCOSH
accuracy 428, 429
algorithm 391
description 291
effect of an argument error 391
error message 517
registers used 439
storage estimate 418
DSIOS error messages
end of data set 500
OPEN statement 501
unformatted I/O 499
DSPAN#/DSPN2#
assembler language requirements 441
description 311
error message 492

storage estimate 421
DSQRT
accuracy 429
algorithm 402
description 293
effect of an argument error 403
error message 515
registers used 439
storage estimate 419
DTAN/DCOTAN
accuracy 428, 429
algorithm 407
description 290
effect of an argument error 408
error message 518
registers used 439
storage estimate 418
DTANH
accuracy 429
algorithm 392
description 291
effect of an argument error 394
registers used 439
storage estimate 419
dummy argument 31
array name 31
definition 546
in a function subprogram 140
in a subroutine subprogram 239
in an ENTRY statement 100
dummy procedure name 138
dump an area of storage 315, 317
DUMP/PDUMP
assembler language requirements 441
format specifications 315
output 315
programming considerations 315
sample printouts 445
storage estimate 421
DUMP/PDUMP subroutine 315
dump, symbolic
See symbolic dump 319
DUPKEY=
REWRITE statement, formatted 228
REWRITE statement, unformatted 230
unformatted, keyed access 265
WRITE statement
formatted, keyed access 255
DVCHK service
assembler language requirements 369
storage estimate 421
DVCHK subroutine 313
DVIOS error messages 485, 500, 504
DYCMN#
description 311
error message 484
storage estimate 421
dynamic common 484

E

E format code 117
EBCDIC codes 361
editing
double precision data 117
integer data 116
real data 117, 118
EJECT statement 94
ELSE IF statement 148
ELSE statement 148
END DEBUG statement 95
description of 95
in debug packet 83
end execution
error messages 529
service subprogram 318
utility subprogram 318
END IF statement 147
end of data set error message 500
end page 112
end program 94
END statement 94
in a function subprogram 95
in a subroutine subprogram 95
end subprogram 94
END=, READ statement
formatted, keyed access 189
formatted, sequential access 193, 207
unformatted, keyed access 200
unformatted, sequential access 203
ENDFILE statement 96
description of 96
invalid 96
valid 96
ENDFILE statement error message 487
ENTRY statement
actual arguments in 100
description of 97
valid 99
equal sign 11
EQUIVALENCE statement 101
description of 101
valid 102
ERF/ERFC
accuracy 429, 430
algorithm 380
description 292
effect of an argument error 381
registers used 439
storage estimate 419
ERR parameters honored for I/O errors 335
ERR=
BACKSPACE statement 67
CLOSE statement 74
DELETE statement 87
ENDFILE statement 96
INQUIRE by file name 155
INQUIRE by unit number 160

OPEN statement 169
READ statement 183, 188
REWIND statement 225
REWRITE statement, formatted 228
REWRITE statement, unformatted 230
WRITE statement 252, 255
ERRMON subroutine 324
error detected 160
error function subprograms
algorithms 380
size 438
error handling subroutines 314, 323
error messages
execution 468
library 464
operator 532
program interrupt 465
error monitor routine 324
error option table 323
error, corrective action after 334
ERRSAV subroutine 325
ERRSET subroutine 325
ERRSTR subroutine 329
ERRTRA subroutine 329
evaluate actual argument 70
examples of numeric format codes 123
executable program
definition 5, 546
names 13
executable statement
definition 6, 546
variable type 25
execute a set of statements 89
execution error messages 468
execution termination 317
execution-time
cautions 349
library 3
EXIST=
INQUIRE by file name 156
INQUIRE by unit number 160
existence of unit 160
existing file
INQUIRE statement 154
OPEN statement 168
existing file, definition of 546
existing unit, definition of 546
EXIT service subprogram
assembler language requirements 441
storage estimate 421
utility subprogram 314
EXIT subroutine 317
EXP
accuracy 430
algorithm 384
description 288
effect of an argument error 386
error message 513
registers used 439
storage estimate 419
underflow 371
explicit type statement
CHARACTER type 103
COMPLEX type 103
DOUBLE PRECISION type 103
INTEGER type 103
LOGICAL type 103
REAL type 103
valid 107
explicitly called subprograms
accuracy statistics 425
algorithms 373
assembler information 438
bit function 441
size 438
exponent overflow exception 313
exponent testing 313
exponential subprograms
algorithms
explicit 386
implicit 412
size 438
exponential subprograms algorithms
implicit 410
exponentiation
explicit
See EXP, QEXP, CQEXP
implicit
with complex base and complex exponent 411
with complex base and integer exponent 312, 411
with integer base and exponent 311, 416
with real base and exponent 312, 413
with real base and integer exponent 311, 415
expression 35
arithmetic 36
character 44
definition 546
evaluation of 35
examples 36
logical 47, 49, 51
relational 45
type of primary in 36
extended error handling 323
extended precision 107
error message 487
subprogram results 436
extensions, IBM, documentation of ix
external 172
file
definition 547
file, sequential 96
function name 97
function, definition 547
I/O unit connected to 172
I/O unit not connected to 172
OPENstatement 172
procedure, definition 5, 547
unit 157
EXTERNAL statement 108
actual argument 108
description of 108

valid 108

F

F format code 116
FCDCD#
algorithm 411
effect of an argument error 411
error message 511
registers used 440
FCDXI#
algorithm 411, 412
description 309
error message 511
registers used 440
storage estimate 419
FCQCQ#
algorithm 411
effect of an argument error 411
registers used 440
FCQXI#
accuracy 430
algorithm 412
description 309
effect of an argument error 413
error message 518
registers used 440
storage estimate 419
FCXPC#
algorithm 411
effect of an argument error 411, 413
error estimate 413
error message 510
registers used 440
storage estimate 419
FCXPI#
algorithm 411
description 309
error message 510
registers used 440
storage estimate 419
FDXPD#
algorithm 413
description 309
effect of an argument error 413
error message 510
registers used 440
storage estimate 419
FDXPI#
algorithm 415
description 309
error message 509
registers used 440
storage estimate 419
file
connected to a unit 156
definition 547
definition statement
definition 547
reference, definition 547
FILE=
INQUIRE by file name 155
OPEN statement 169
FILEOPT 319
first character of record 111
fix 64
fixed-form source statement
comments 7, 76
continuation line 8
example of 8
identification 8
initial line 8
label 7
number 236
FIXPI#
algorithm 416
description 309
error message 508
registers used 440
storage estimate 419
flagger, source language 347
float 64
floating-point registers 441
FMT=
READ statement 182, 186
REWRITE statement, formatted 227
WRITE statement 250, 254
FORM=
INQUIRE by file name 157
INQUIRE by unit number 162
OPEN statement 170
format
identifier 250, 254
identifier, READ 182, 186
notation
format codes
begin data transmission (T) 128
blanks, interpretation of 130, 131
character constant transmission (H) 127
character data transmission (A) 125
character skipping (X) 128
colon 132
double precision data editing (Q) 117
format specification reading 133
general rules 111
group format specification 129
hexadecimal data transmission (Z) 122
integer data editing (I) 116
list-directed 134
logical variable transmission (L) 125
numeric 123
plus character control (S, SP, SS) 130
real data 116, 117
real data editing 118
scale factor specification (P) 120
slash 132
format control 112
FORMAT statement

A code 125
BN code 130
BZ code 131
colon code 132
D code 117
definition 108
E code 117
examples 123
F code 116
format specification reading 133
forms of 114
G code 118
general rules for conversion 111
group format specification 129
H code 127
I code 116
L code 125
list-directed formatting 134
numeric code 123
P code 120
Q code 117
S code 130
slash code 132
SP code 130
SS code 130
T code 128
X code 128
Z code 122

formatted
- input/output
  - INQUIRE statement 156, 161
- PRINT statement 175
- READ statement
  - direct access 182
  - keyed access 186
  - sequential access 192
- record 111
  - definition 547
  - INQUIRE statement 156
  - OPEN statement 170
- WRITE statement
  - direct access 250
  - keyed access 254
  - sequential access 258

FORMATTED=
- INQUIRE by file name 156
- INQUIRE by unit number 161

forms of a FORMAT statement 114

FORTRAN-supplied procedure 13
- See also intrinsic function
- identified 13
- keywords 13

FQXPI#
- accuracy 430
- algorithm 416
- description 309
- effect of an argument error 416
- error message 511
- registers used 440
- storage estimate 419

FQXPQ#
- accuracy 430
- algorithm 414
- description 309
- effect of an argument error 415
- error message 512
- registers used 440
- storage estimate 419

FQXP2#
- accuracy 430
- algorithm 416
- description 309
- error message 515
- registers used 440
- storage estimate 419

free-form source statement
- comments 9, 76
- continuation line 10
- continued line 9
- example of 10
- initial line 9
- maximum length 10
- minus sign 9
- statement label 9
- statement number 236

FRXPI#
- algorithm 415
- description 309
- error message 509
- registers used 440
- storage estimate 419

FRXPR#
- accuracy 430
- algorithm 413
- description 309
- effect of an argument error 413
- error estimate 413
- error message 510
- registers used 440
- storage estimate 419

function
- definition 547
- evaluating 35
- reference 233

FUNCTION statement 137, 233

function subprogram 56
- actual arguments 140
- definition 547
- description of 56
- dummy arguments 140
- END statement 95
- ENTRY statement 97
- RETURN statement 222

G

G format code 118
gamma subprograms
  algorithms 387
  size 438
GAMMA/ALGAMA
  accuracy 427, 430
  algorithm 387
  description 293
  effect of an argument error 389
  error message 525
  registers used 439
  storage estimate 417
generic names 164
generic names for intrinsic functions 301
glossary 545
GO TO statement
  assigned 143
  computed 144
  description of 142
  unconditional 144
group format
  nesting 113
  specification 129

H

H format code 127
hexadecimal
  constant 23, 547
  data transmission 122
hierarchy of operations
  arithmetic 37
  arithmetic operators 49
  character operators 49
  definition 547
Hollerith constant 22
  definition 22
  valid 23
hyperbolic cosine subprograms
  algorithms 390
  size 438
  storage estimate 417
hyperbolic sine subprograms
  algorithms 390
  storage estimate 417
hyperbolic tangent subprograms
  algorithms 392
  size 438
  storage estimate 417

I

I format code 116
I/O
  definition 547
  list omitted from READ or WRITE 112
  list-directed READ statement
    from external devices 210
    with internal files 213
  list-directed WRITE statement
    to external devices 274
    with internal files 278
  list, definition 547
I/O errors, IOSTAT and ERR parameters honored for 335
IABS
  See ABS/IABS/DABS/QABS
IAND
  description 300
  storage estimate 422
IBCLR
  description 300
  storage estimate 422
IBCOM error message 496, 530
IBM extensions, documentation of ix
IBM FORTRAN features 351
IBSET
  description 300
  storage estimate 422
ICHAR
  description 299
  storage estimate 421
ID=
  READ statement 179
  WAIT statement 244
  WRITE statement 247
identification, fixed-form 8
identify
  function subprogram 137
  statements 236
  user-supplied subprogram 108
IDIM
  description 295
  registers used 439
  storage estimate 418
IDNINT/IFIX/INT
  description 295
  registers used 439
  storage estimate 417
IEOR
  description 300
  storage estimate 422
IF statement
  arithmetic 145
  block 146
  description of 145
  logical 150
IF-block 146
IF-level 146

IMPLICIT type statement 151
implicitly called subprograms
  assembler information 440
  description 309
  name generation 409
implied DO
  definition 547
  in DATA statement 91
  in PRINT statement 92
  in READ statement 92
  in WRITE statement 92
implied DO error message 496
INCLUDE statement 153
INDEX
  description 299
  error message 493
  storage estimate 421
industry standards iv
information about file 154
inherited length of character string 105
INIT
  DEBUG statement 82
initial line 8, 9
  fixed-form 8
  free-form 9
input data, NAMELIST statement 167
input-output statement error messages 486, 498, 499
input/output statement 55
  BACKSPACE 67
  CLOSE 74
  description of 55
  ENDFILE 96
  FORMAT 108
  implied DO 92
  INQUIRE 154
  OPEN 168
  PRINT 175
  READ 178
  REWIND 225
  WAIT 244
  WRITE 246
input/output statement error messages 485, 498, 499
input/output unit 172
  connected to external file 172
  not connected to external file 172
  OPEN statement 172
  PRINT statement 92
  READ statement 92
  WRITE statement 92
INQUIRE statement
  by file name 155
  by unit number 159
insert statements 153
integer constant 16
  definition 16, 547
  invalid 16
  subscripts and substrings 101
  valid 16
integer data editing 116
integer expression
  arithmetic expressions 36
  definition 547
  subscripts and substrings 101
integer type 151
  definition 547
  explicit type statement 103
  IMPLICIT type statement 151
INTEGER type statement 103
integer value error message 497
integer variable
  READ statement 183, 187
  storage length 25
  WRITE statement 251, 254
integer-to-integer 309
internal file
  definition 547
  reading sequentially 206
  writing sequentially 270
internal file error message 495, 496
interruption procedures 463
intrinsic function
  definition 5, 547
  INTRINSIC statement 164
intrinsic functions, generic names for 301
INTRINSIC statement 164
invalid VS FORTRAN programs 4
IOR
  description 300
  storage estimate 422
IOSTAT parameters honored for I/O errors 335
IOSTAT=
  BACKSPACE statement 67
  CLOSE statement 74
  DELETE statement 87
  ENDFILE statement 96
  INQUIRE by file name 155
  INQUIRE by unit number 160
  OPEN statement 170
  READ statement 184, 188
  REWIND statement 225
  REWRITE statement, formatted 228
  REWRITE statement, unformatted 230
  WRITE statement 252, 255
ISCII/ASCII codes 361
ISHFT
  description 300
  storage estimate 422
ISIGN
  description 295
  registers used 439
  storage estimate 419
ISO definitions 545

K

key sequenced data set error message 486
KEY=, READ statement
  formatted, keyed access 188, 199
keyed access
  input/output

INQUIRE statement 156
READ statement
formatted input 186
unformatted input 198
KEYED=
INQUIRE by file name 156
INQUIRE by unit number 161
KEYEND=
INQUIRE by file name 158
INQUIRE by unit number 163
KEYGE=, READ statement
formatted, keyed access 188, 199
KEYGT=, READ statement
formatted, keyed access 188, 199
KEYID=
INQUIRE by file name 158
INQUIRE by unit number 163
KEYID=, READ statement
formatted, keyed access 188
unformatted, keyed access 199
KEYLENGTH=
INQUIRE by file name 158
INQUIRE by unit number 163
KEYS=, OPEN statement 171
KEYSTART=
INQUIRE by file name 158
INQUIRE by unit number 163
keywords 13

L

L format code 125
labeled common
See named common
LANGLVL(66) features 358
LANGLVL(77) features 351
language syntax 6
LASTKEY=
INQUIRE by file name 159
INQUIRE by unit number 163
LASTRECL=
INQUIRE by file name 159
INQUIRE by unit number 163
LDFIO error messages 497, 503, 506
leading blanks 113
LEN
description 299
storage estimate 421
length of character string, inherited 105
length specification
definition 547
IMPLICIT type statement 151
letter
definition 547
source statement characters 11
lexical compare error message 493
LGE
description 299
storage estimate 421
LGT
description 299
storage estimate 421
library
availability 433
contents 3
error procedures 463
execution-time routine storage estimates 424
interruption procedures 463
messages 464
procedure messages 463
list-directed
definition 547
formatting 134
I/O
reading from external devices 210
reading internally 213
writing internally 278
writing to external devices 274
PRINT statement 175
literal parameter error message 489
LLE
description 299
storage estimate 421
LLT
description 299
storage estimate 421
load module execution termination 317
log-gamma subprograms
algorithms 389
registers used 439
LOG/LOG10
See ALOG or DLOG
logarithmic subprograms
algorithms 394
logical assignment statement 61
logical constant 21
definition 21, 547
logical expression
definition 547
invalid 49
order of computations in 49
use of parentheses in 51
valid 48
logical IF statement 150
logical operation 52
type and length of the result 52
logical operator 47
AND 48
definition 547
description of 47
EQV 48
examples 48
invalid 48
NEQV 48
NOT 48
OR 48
valid 48
logical primary

See primary
logical type 103, 151
LOGICAL type statement 103
explicit type statement 103
primary, definition 547
type, definition 547
logical unit number error message 489
logical variable
storage length 25
transmission 125
looping
definition 547
when using DEBUG 83
lower dimension bound
DIMENSION statement 88
explicit statement 105
LTNCT
error message 518
LXCMP error message 493

M

main program
assign a name to 177
definition 5, 547
PROGRAM statement 177
statement (PROGRAM) 55
mathematical exception tests 313
mathematical library subroutines
accuracy figures 425-432
algorithms 371
control of program exceptions 371
explicitly called 438
implicitly called 309
performance statistics 425-432
result registers 438
results 436
size 438
storage estimate 417
use in assembler language 433
use in VS FORTRAN 307
mathematical subroutine errors 341
maximum
record size 111
statement length, free-form 10
maximum/minimum functions 296
messages
execution error 468
library 464
operator 532
program interrupt 465
minus sign 11
MOD
description 294
registers used 439
storage estimate 419
modify
block size 318
buffer offset 318
modular arithmetic subprograms
size 438
module names
bit 533
character 533
mathematical 533
reentrant 533
multiphase job running 318
multiply complex numbers 309

N

name 11
array 77, 87
block of data 69
CALL statement 97
constant 113
definition 11, 548
description of 11
elements of a program 11
file 155, 156
function reference 97
generic 164
specific 164
unit 161
variable 77
NAME=
INQUIRE by file name 156
INQUIRE by unit number 161
named common 78
blank common blocks 78
definition 78, 548
description of 78
NAMED=
INQUIRE by file name 156
INQUIRE by unit number 161
NAMEL error messages 502
NAMELIST name
in READ statement
with external devices 216
with internal files 219
in WRITE statement
with external devices 280
with internal files 282
NAMELIST statement
input data 167
output data 168
NAMELIST statement error messages 502
names in READ and WRITE statements 166
natural logarithmic subprograms
algorithms 394
size 438
nested DO
definition 548
example of 91
nesting of group formats 113
new file 169
new page 112
NEXTREC=

INQUIRE by file name 157
INQUIRE by unit number 162
NINT
description 295
registers used 439
storage estimate 417
nonexecutable statement
definition 6, 548
nonexisting file
definition 548
nonmathematical arguments 313
NOT
description 300
storage estimate 422
NOTFOUND=
formatted, keyed access 189
unformatted, keyed access 200
null 157, 162
NUM=
READ statement
direct access 196
keyed access 200
sequential access 203
WAIT statement 245
WRITE statement
direct access 262
keyed access 265
sequential access 268
number
fixed-form 236
free-form 236
last record 157, 162
statement 155, 236
NUMBER=
INQUIRE by file name 157
INQUIRE by unit number 162
numeric
constant 16
definition 548
data format codes 113
format code 123
numeric character
See arithmetic constant

O

object-time dimensions 31, 78
old file 169
OPEN statement 168
OPEN statement error message 489
OPENED=
INQUIRE by file name 156
INQUIRE by unit number 161
operator messages 532
OPSYS
description 318
error messages 489
OPSYS subroutine 318
option
default 328, 334
in DEBUG statement 82
option table
default values 334
entry 332
option table, error 323
order of
computation 49
statements 58
output data, NAMELIST statement 168
OVERFL
description 313
storage estimate 421
OVERFL subroutine 313
overflow
algorithms 371
error message 465
indicator service subprogram 313
terminal 371

P

P format code 120
page control 112
PARAMETER statement 173
parameters, correct 435
parentheses error message 485
PASSWORD=, OPEN statement 171
PAUSE statement 175
PAUSE statement error message 532
performance statistics 425
period 11
plus character control 130
plus sign 11
position an external file 225
preconnected file
definition 6, 548
formatted READ—direct access 184
formatted READ—sequential access 194
formatted WRITE—sequential access 260
READ with list-directed I/O 211
READ with NAMELIST 217
unformatted READ—direct access 197
unformatted READ—sequential access 204
unformatted WRITE—direct access 263
WRITE with list-directed I/O 275
WRITE with NAMELIST 281
predefined specification
definition 548
specify variable 27
preserving a minus sign
free-form 9
primary 36
definition 548
description of 36
logical 47
print control 111

PRINT statement 175
  description of 175
  implied DO in 92
procedure
  BLOCK DATA 56
  definition 5, 548
  dummy 98, 100, 138
  subprogram
    definition 548
    description of 56
program exceptions 370
PROGRAM statement 55, 177
program unit
  definition 6, 548
  order of statements in 57
program-interrupt messages 341, 465
programming considerations
  CDUMP/CPDUMP 317
  DUMP/PDUMP 316
  SDUMP 320

Q

Q format code 117
QABS
  description 292
  registers used 439
  See ABS/IABS/DABS/QABS
  storage estimate 417
QARCOS
  See QARSIN/QARCOS
QARSIN/QARCOS
  accuracy 430
  algorithm 375
  description 290
  effect of an argument error 375
  error message 527
  registers used 439
  storage estimate 419
QATAN/QATAN2
  accuracy 430
  algorithm 378
  description 290
  effect of an argument error 380
  error message 527
  registers used 439
  storage estimate 418, 419
QCONJG
  description 294
  size 438
  storage estimate 418
QCOS
  See QSIN/QCOS
QCOSH
  See QSINH/QCOSH
QCOTAN
  See QTAN/QCOTAN
QDIM
  description 295
  registers used 439
QERF/QERFC
  accuracy 431
  algorithm 382
  description 292
  effect of an argument error 384
  size 439
  storage estimate 419
QEXP
  accuracy 431
  algorithm 386
  description 288
  effect of an argument error 387
  error message 526
  size 439
  storage estimate 418, 419
QIMAG
  description 294
  registers used 439
  storage estimate 417
QINT
  description 294
  registers used 439
  storage estimate 417
QLOG/QLOG10
  accuracy 431
  algorithm 396
  description 288
  effect of an argument error 397
  error message 526
  size 439
  storage estimate 419
QMOD
  description 294
  registers used 439
  storage estimate 417
QP extend 64
QP float 64
QSIGN
  description 295
  registers used 439
QSIN/QCOS
  accuracy 430, 431
  algorithm 400
  description 290
  effect of an argument error 400
  error message 527
  registers used 439
  storage estimate 418, 419
QSINH/QCOSH
  accuracy 430, 431
  algorithm 392
  description 291
  effect of an argument error 392
  error message 527
  registers used 439
  storage estimate 419
QSQRT
  accuracy 431
  algorithm 404
  description 293

effect of an argument error 404
error message 525
registers used 439
storage estimate 420
QTAN/QCOTAN
accuracy 431
algorithm 408
description 290
effect of an argument error 408
error message 528
registers used 439
storage estimate 419
QTANH
accuracy 431
algorithm 393
description 291
effect of an argument error 394
registers used 439
storage estimate 420
quotation mark 11

R

range of a DO
definition 548
range of an implied DO 92
READ statement
asynchronous 179
description of 178
formatted with direct access 182
formatted with keyed access 186
formatted with sequential access
external devices 192
internal files 206
forms of 178
implied DO in 92
unformatted with direct access 196
unformatted with keyed access 198
unformatted with sequential access 203
with list-directed I/O
external devices 210
internal files 213
with NAMELIST
external devices 216
internal files 219
READ statement error message 496
READ=
INQUIRE by file name 158
INQUIRE by unit number 163
reading format specifications 133
READWRITE=
INQUIRE by file name 158
INQUIRE by unit number 163
real assign 64
real constant 17
definition 17, 548
invalid 19
valid 18
real data
editing 117, 118
transmission 116
real data of length 8
See double precision
real type 103, 151
definition 548
explicit type statement 103
IMPLICIT type statement 151
REAL type statement 103
real variable, storage length 25
REAL*16
See double precision 107
REAL*8
See double precision
real-to-integer 309
real-to-real 309
REC=
READ statement 183
WRITE statement 251
RECL=
INQUIRE by file name 157
INQUIRE by unit number 162
OPEN statement 170
record
definition 548
FORMAT statement 108
length 157, 162, 170
number of last 157, 162
record length error message 486
relational expression 45
definition 548
description of 45
invalid 47
length of 45
valid 46
relational operator 45
definition 548
description of 45
equal to 45
greater than 45
greater than or equal to 45
less than 45
less than or equal to 45
not equal to 45
replace value of expression 60
reposition a file 67
request traceback 329
required order of statements 58
retain definition status 232
return control to calling program 222
RETURN statement
description of 222
in a function subprogram 222
in a subroutine subprogram 223
REWIND statement 225
REWRITE statement, formatted 227
REWRITE statement, unformatted 230
rewriting records 227
rules for data conversion 111

run multiphase jobs 318

S

S format code 130
sample storage printout 445
save areas 442
save option table entry 325
SAVE statement 232
scale factor
    definition 548
    specification 120
scratch a file 169
SDUMP
    format specifications 320, 446
    output 319
    programming considerations 320
    sample printouts 445
    storage estimate 421
SDUMP subroutine
    description 319
    output of symbolic dump 452
sequential access
    input/output
        INQUIRE statement 156, 161
    READ statement 192, 203
    WRITE statement 258, 268
sequential access file 483
SEQUENTIAL=
    INQUIRE by file name 156
    INQUIRE by unit number 161
service subprograms
    assembler information 434
    CDUMP/CPDUMP 317
    DUMP/PDUMP 315
    end execution 317
    EXIT 317
    mathematical exception test 313
    OPSYS 318
    SDUMP 319
    sizes 434
    XUFLOW 322
service subroutines 313
share storage 77, 101
SIGN
    description 295
    registers used 439
    storage estimate 419
SIN/COS
    accuracy 427, 432
    algorithm 398
    description 290
    effect of an argument error 399
    error message 513
    registers used 439
    storage estimate 418, 420
sine subprograms
    algorithms 398
    size 438
SINH/COSH
    accuracy 427, 432
    algorithm 390
    description 291
    effect of an argument error 390
    error message 514
    registers used 439
    storage estimate 418
skip a line 112
skipping characters 128
slash 11
slash format code 132
source language flagger 347
source language statement
    fixed-form 7
    free-form 9, 10
source statement characters 10
    description of 10
    digit 11
    letter 11
    special characters 11
SP format code 130
special character set 11
special characters
    parentheses 11
specific names 164
specification
    subprogram
        definition 548
specification statement 55
    CHARACTER type 103
    COMMON 77
    COMPLEX type 103
    definition 548
    DIMENSION 87
    DOUBLE PRECISION type 103
    EQUIVALENCE 101
    explicit type 103
    EXTERNAL 108
    IMPLICIT type 151
    INTEGER type 103
    INTRINSIC 164
    LOGICAL type 103
    NAMELIST 166
    PARAMETER 173
    REAL type 103
    SAVE 232
SQRT
    accuracy 432
    algorithm 402
    description 293
    effect of an argument error 402
    error message 512
    registers used 439
    storage estimate 418, 420
square root subprograms
    algorithms 402
    examples 443
    size 438
SS format code 130

start
a new page 94
display 243
statement
definition 548
descriptions 53
fixed-form number 7, 236
free-form label 9
free-form number 236
function definition, definition 548
function reference, definition 549
function statement 233
function, definition 548
label 13
label, definition 549
label, fixed-form 7
number 59, 236
number, definition 549
READ statement 182, 187
WRITE statement 250, 254
statement function
statement 233
STATUS=
CLOSE statement 74
OPEN statement 169
stop display 243
stop program 94
STOP statement 237
STOP statement error message 532
storage dump service
subprograms 315
storage estimates
bit subprograms 422
character subprograms 421
execution-time routines 424
extended precision routines 434
library execution-time routines 424
mathematical subprograms 417-420
service subprograms 421
storage printout, sample 445
store entry in option table 329
SUBCHK
DEBUG statement 82
SUBCHK function of DEBUG 84
subprogram
BLOCK DATA statement 56, 69
definition 5, 549
ENTRY statement 97
FUNCTION statement 56, 137
RETURN statement 222
SAVE statement 232
statement function statement 233
SUBROUTINE statement 56, 238
subprograms
character 310
implicit 309
service 313-321
subprograms, explicitly called
bit manipulation 300
character manipulation 299
conversion 296
maximum/minimum 296
SUBROUTINE statement 238
subroutine subprogram
actual arguments 239
definition 549
description of 56
dummy arguments 239
END statement 95
ENTRY statement 97
naming 56
RETURN statement 223
service and utility subroutines 313
subscript 28
definition 549
identify array element 28
in DATA statement 80
quantity, definition 549
substring 32
description of 32
expression 32
in DATA statement 80
SUBTRACE
DEBUG statement 83
symbolic dump
how to call 319
symbolic name
See name
syntax 6

T

T format code 128
table, error option 323
TAN/COTAN
accuracy 427, 432
algorithm 406
description 290
effect of an argument error 407
error message 515
registers used 439
storage estimate 418
tangent subprograms
algorithms 406
registers used 439
TANH
accuracy 432
algorithm 392
description 291
effect of an argument error 392
registers used 439
storage estimate 420
terminate
execution 237
program 94
the last debug packet 95
terminate execution (EXIT) 317
terminate execution of load module 317

test
    exponents 313
    for divide-check exception 313
    values 89
TRACE
    DEBUG statement 82
TRACE OFF statement 243
    description of 243
    in debug packet 83
TRACE ON statement 243
    description of 243
    in debug packet 83
traceback request 329
transfer control
    to statement number 142
    to subroutine subprogram 70
transmission
    character constants 127
    character data 125
    hexadecimal data 122
    logical variables 125
trigonometric subprograms
    algorithms 373
    error messages 513
    size 420
truncation subprograms
    registers used 439
    storage estimate 417
two-to-real 309
type declaration
    by EXPLICIT type statement 28
    by IMPLICIT statement 27
    declaration of an array 30
    definition 549
    predefined 27
type specification 151

U

unary minus 38
unary plus 38
unconditional GO TO statement 144
underflow
    algorithms 370
    error message 466
    terminal 371
unformatted
    input/output
        INQUIRE statement 157, 161
    OPEN statement 170
    READ statement
        direct access 196
        keyed access 198
        sequential access 203
    record
        definition 549
        INQUIRE statement 157
        OPEN statement 170
    WRITE statement
        direct access 262
        keyed access 265
        sequential access 268
unformatted I/O 499
UNFORMATTED=
    INQUIRE by file name 157
    INQUIRE by unit number 161
unit
    connected 161
    connected to external file 172
    DEBUG statement 82
    definition 549
    identifier, definition 549
    INQUIRE statement 160
    not connected to external file 172
    number 160, 169
    OPEN statement 169
UNIT=
    BACKSPACE statement 67
    CLOSE statement 74
    DELETE statement 87
    ENDFILE statement 96
    INQUIRE by unit number 160
    OPEN statement 169
    READ statement 179
    REWIND statement 225
    REWRITE statement, formatted 227
    REWRITE statement, unformatted 230
    WAIT statement 244
    WRITE statement 247
unknown file 169
updating records 227
upper dimension bound 30
    DIMENSION statement 88
    explicit statement 105
user exit routine, coding the 328
utility service subprograms
    CDUMP/CPDUMP 317
    DUMP/PDUMP 315
    EXIT 317
    OPSYS 318
    SDUMP 319
    XUFLOW 322
utility subroutines 313

V

valid VS FORTRAN programs 4
variable
    character 32
    definition 549
    description of 24
    names, invalid 25
    names, valid 24
    types and lengths of 25
VASYN error messages
    addressing incorrect 504
    asynchronous I/O not supported 485
    asynchronous I/O subtask 497

blocksize not specified 507
end of data set 500
end of record 498
OPEN failed 501
REWIND statement 524
unformatted I/O 499
WAIT statement 525
VCLOS error messages 492
VCOMH error messages 485, 498
VCTVH error messages
illegal decimal character 499
illegal hexadecimal character 503
illegal integer value 497
illegal value 503
VDIOS error messages 488, 501
VERRM error messages 530
VFILOPT 319
VFNTH error messages 465
VIIOS error messages 495
VINQR error messages 490
VIOLP error messages 498
VMOPT error messages 530
VOPEN error messages 483, 490
VS FORTRAN statements 53
VSAM error messages 485
VSCOM error messages 483
VSCOM#
subprogram 437
VSERH error messages 504
VSIOS error messages
BACKSPACE statement 499
end of data set 500
file is unusable 482
OPEN statement 501
OPEN/CLOSE 488
REWIND statement 524
sequential I/O 504
unformatted I/O 499
VSTAE error messages 507
VVIOS error messages 485, 500, 504

W

WAIT statement 244
write an end-of-file record 96
WRITE statement
asynchronous 247
formatted with direct access 250
formatted with keyed access 254
formatted with sequential access
external devices 258
internal files 270
forms of 246
implied DO in 92
unformatted with direct access 262
unformatted with keyed access 265
unformatted with sequential access 268
with list-directed I/O
external devices 274
internal files 278
with NAMELIST
external devices 280
internal files 282
WRITE statement error message 485
WRITE=
INQUIRE by file name 158
INQUIRE by unit number 163

X

X format code 128
XUFLOW
storage estimate 421
XUFLOW subroutine
description 322

Z

Z format code 122
zero 157, 162

VS FORTRAN
Language and Library Reference
SC26-4119-0

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ____________________

Previous TNL ____________________

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

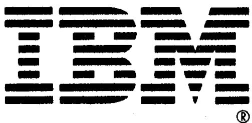
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | Please do not staple | Fold and tape



VS FORTRAN
Language and Library Reference
SC26-4119-0

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ________________

Previous TNL ________________

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SC26-4119-0

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

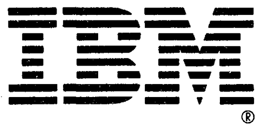
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | Please do not staple | Fold and tape



VS FORTRAN Language and Library Reference (File No. S370-40) Printed in U.S.A. SC26-4119-0

VS FORTRAN
Language and
Library Reference

File No. S370-40

SC26-4119-0

Printed in U.S.A.

IBM